Institut für Informatik
Arbeitsgruppe Softwaretechnik
Warburger Straße 100
33098 Paderborn

Inkrementelle Entwurfsmustererkennung

Schriftliche Arbeit
zur Erlangung des Grades
"Doktor der Naturwissenschaften"

vorgelegt von

Dipl.-Inform. Jörg Niere
Wenschtstr. 30
57078 Siegen

Paderborn, Juni 2004

Zusammenfassung

In den letzten 10 Jahren ist die Größe der Softwaresysteme drastisch gestiegen. Aufgrund der geringeren Kosten sind Änderungen an den Systemen lediglich in der Implementierung vorgenommen worden, ohne die Dokumentation entsprechend anzupassen. Die Wartung solcher Systeme ist durch die nicht vorhandene oder teilweise fehlerhafte Dokumentation aufwändig. Zudem haben Änderungen vielfach unerwartete Nebeneffekte, was die Kosten für die Änderungen zusätzlich erhöht. Eine Rückgewinnung der Dokumentation aus der Implementierung des Softwaresystems ist daher notwendig.

Bei der Analyse der Implementierung von Softwaresystemen zur Rückgewinnung der Dokumentation hat sich herausgestellt, dass gleichartige Implementierungen für wiederkehrende Probleme vorhanden sind. Diese Probleme mit entsprechenden Lösungen sind als Entwurfsmuster bekannt und werden unter anderem zur Dokumentation eines Softwaresystems eingesetzt. Die Beschreibung eines Entwurfsmusters ist dabei informal, wodurch zu einem Entwurfsmuster eine hohe Anzahl Implementierungsvarianten existieren. Automatische Analysen zur Erkennung von Entwurfsmusterinstanzen sind aufgrund der hohen Anzahl Implementierungsvarianten eines Entwurfsmusters in großen Systemen mit hundert tausend oder millionen Zeilen Quelltext nicht geeignet.

In dieser Arbeit wird eine inkrementelle Entwurfsmustererkennung zur Rückgewinnung der Dokumentation eines Softwaresystems auf Basis von Graphtransformationsregeln in Kombination mit Fuzzymengen vorgestellt. Durch Angabe von Genauigkeitswerten kann die Präzision einer Regel angegeben werden, die in die Ergebnisse der automatischen Analyse einfließen. Ein inkrementeller Regelausführungsmechanismus produziert frühzeitig relevante Ergebnisse, so dass insbesondere bei der Analyse großer Softwaresysteme gegebenenfalls Regeln aufgrund der Ergebnisse frühzeitig angepasst werden können. Außerdem können Hypothesen und Informationen zusätzlich zum Quelltext mit in die Analyse eingebracht werden.

Der vorgestellte Ansatz ist dabei nicht auf die Erkennung von Entwurfsmustern beschränkt, sondern lässt sich auch für die Erkennung von Implementierungsmustern, Verteilungsmustern, Architekturmustern oder Mustern aus Mustersprachen verwenden.

Danksagung

Mein Dank geht an Wilhelm Schäfer für die wissenschaftliche Betreuung und die vielen Diskussionen in den letzen Jahren sowie die Unterstützung bei der Anfertigung dieser Arbeit. Gregor Engels danke ich für die Übernahme des Koreferats und die Gespräche während unserer Kaffeerunden.

Dank an Albert Zündorf für die frühzeitige Veröffentlichung seines formalen Regelwerks, ohne das diese Arbeit wahrscheinlich wesentlich umfangreicher geworden wäre. Außerdem sind viele unserer kreativen Gedanken und Ideen in Fujaba und in dieser Arbeit verwirklicht worden.

Weiterhin danke ich den (ehemaligen) Mitgliedern unserer Arbeitsgruppe Jens H. Jahnke, Ulrich Nickel, Holger Giese, Ekkart Kindler, Matthias Gehrke, Sven Burmester, Daniela Schilling, Lothar Wendehals, Robert Wagner, Björn Axenath, Jörg P. Wadsack, Matthias Tichy und Matthias Meyer für viele Diskussionen, kuriose Ideen und das Korrektur lesen dieser Arbeit. Das kreative Arbeitsklima ist dabei durch viele Fußballspiele, Kaffeestunden, Meilensteinfeiern, Feierabendbiere und andere Freizeitgestaltungen gefördert worden. Dank auch an Jutta Haupt für die Überwindung mancher bürokratischer Hürde und Jürgen Maniera für die technische Unterstützung.

Für die technische Realisierung gehört mein Dank dem Fujaba-Team mit mehr als hundert registrierten Nutzern und Entwicklern. Insbesondere geht mein Dank an die Studenten, die an der Entwicklung des Prototypen als wissenschaftliche Hilfskräfte oder durch Studien- oder Diplomarbeiten beteiligt waren.

Mein ganz besonderer Dank geht an meine Frau Regine und unsere Kinder Niklas und Kim Julia. Sie haben mich vor allem immer wieder seelisch aufgebaut und mich bei der Fertigstellung dieser Arbeit sehr unterstützt. Vielen Dank auch an meine Eltern und all die anderen Babysitter, die das ein oder andere Mal mir ein paar Stunden zum Schreiben und uns ein paar Stunden zum Ausspannen ermöglicht haben.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

KAPITEL 1: EINLEITUNG

"Never touch a running system." ist eines der meist verwendeten Sprichwörter im Bereich der Informatik. Es beschreibt die Tatsache, dass schon kleine Änderungen an einem Teilsystem unerwünschtes beziehungsweise nicht eingeplantes Verhalten des Gesamtsystems hervorrufen. Systeme, in denen solche Probleme auftreten, sind das Produkt des in den letzten Jahren immer schneller wachsenden IT-Markts und der damit verbundenen immer schnelleren Produktion und Anpassung der Software.

Insbesondere durch schnelle, kostengünstige Anpassungen der Software entstehen wenig dokumentierte Systeme, die meist aus mehreren hunderttausend oder millionen Zeilen Quelltext bestehen. Sie sind durch mehrere Generationen von Entwicklern weiter entwickelt worden und gehören damit zu den so genannten *Legacysystemen*. In Legacysystemen werden Änderungen oder Erweiterungen in vielen Fällen nur in der Implementierung des Systems vorgenommen; die Dokumentation wird aber nicht entsprechend angepasst.

Änderungsauswirkungen sind aufgrund der nicht mehr aktuellen Dokumentation des Systems nicht mehr abzuschätzen. Daraus folgt, dass mit zunehmender Anzahl von Änderungen an einem System die Anzahl und Komplexität der entstehenden Nebeneffekte steigt. Eine Vermeidung beziehungsweise Reduzierung der Nebeneffekte ist nur durch eine aktuelle Dokumentation eines Softwaresystems zu erreichen. Im Falle von Legacysystemen, bei denen Änderungen lediglich in der Implementierung vorgenommen worden sind, muss die Dokumentation aus dem System wiederhergestellt werden [MJS+00] (englisch: reverse-engineering). Werden anschließend Änderungen am System vorgenommen, so wird die Kombination aus *Reverse-Engineering* und anschließender Weiterentwicklung als *Reengineering* bezeichnet.

Die Kosten für das Reverse-Engineering eines Softwaresystems machen den größten Teil der Wartungskosten aus, wobei die Wartungskosten nach Schätzungen etwa 50%-80% der Gesamtkosten betragen [Mül97, Bas90, ZSG79]. Die Kosten für eine sich anschließende Weiterentwicklung eines Systems sind vergleichbar gering, so dass die Kosten für ein Reengineering im Wesentlichen durch die Kosten des Reverse-Engineering bestimmt sind. Kostenintensive Reengineering-Beispiele der letzten Jahre waren das 99er Problem, bei dem die Jahreszahl 99 als Dateiendmarkierung interpretiert worden ist sowie das Jahr 2000 Problem und die Währungsumstellung in Europa 2001. Insbesondere die Umstellung der Softwaresysteme im Hinblick auf das Jahr 2000 hat mehrere 100 Milliarden US Dollar gekostet. Trotz dieser hohen Investitionen konnte nicht garantiert werden, dass alle Softwaresysteme am 01.01.2000 korrekt arbeiten würden. So wurden zum Beispiel si-

cherheitskritische Systeme oder Finanzsysteme, soweit möglich, zur Jahreswende abgeschaltet und kontrolliert wieder hochgefahren.

Die hohen Kosten für das Reverse-Engineering eines Softwaresystems resultieren aus der Tatsache, dass die Dokumente einerseits nur durch eine Analyse der Implementierung zurückgewonnen werden können, andererseits eine automatische Analyse nur für Teilbereiche des Reverse-Engineerings erfolgreich ist. Hinzu kommt, dass automatische Analysen in Bezug auf große Systeme vielfach versagen. Rein manuelle Analysen sind sehr kostenintensiv. Die Alternativen sind semi-automatische Analysen, bei denen das Reverse-Engineering teilweise manuell und teilweise automatisch durchgeführt wird.

Der Vorteil bei semi-automatischen Analysen ist, dass das Wissen und die Erfahrung der Person, die das Reverse-Engineering durchführt (englisch: Reengineer), in die automatische Analyse einfließen kann. Ein *Reengineer* kann zum Beispiel Teildokumentationen oder Informationen aus Gesprächen mit den Entwicklern des Systems mit in die Analyse einbringen. Andererseits profitiert der Reengineer durch eine gezielte automatische Analyse, bei der Ergebnisse schnell und präzise produziert werden können. Er kann dadurch frühzeitig auf möglicherweise fehlerhafte Ergebnisse oder eine fehlschlagende Analyse reagieren und somit die automatische Analyse steuern.

1.1 Entwurfsmuster in Softwaresystemen

Bei der Analyse eines Softwaresystems ist festzustellen, dass bestimmte Teile mehrfach vorhanden sind, sich aber fast immer unterscheiden. Diese *Muster* charakterisiert Christopher Alexander mit:

> *"Each pattern describes a problem which occurs over and over again in our environments, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."*
>
> [AIS+77]

Ursprünglich hat sich diese Beschreibung auf Bauwerke und ihre Architektur bezogen. Sie trifft aber ebenfalls auf die Softwareentwicklung und insbesondere auf die objektorientierte Softwareentwicklung zu. Muster werden aufgrund ihres Abstraktionsgrades und ihrer Anwendung unterschieden. So existieren zum Beispiel *Businessmuster*, *Prozessmuster* [Amb98] und für den Bereich der Softwareentwicklung *Architekturmuster* [BMR+96] und *Implementierungsmuster* [Lan99]. *Entwurfsmuster* (englisch: Design-Pattern) repräsentieren Lösungen für häufig wiederkehrende Designprobleme, insbesondere in der objektorientierten Softwareentwicklung in Verbindung mit Modellierungssprachen wie der UML [UML].

Gamma et al. stellen in ihrem Buch "Design-Patterns: Elements of Reusable Objekt-Oriented Software" [GHJV95] eine Reihe von Entwurfsmustern vor. Die vorgestellten Entwurfsmuster sind das Ergebnis eines Reverse-Engineering von Softwaresystemen bei

IBM. Die untersuchten Softwaresysteme sind beziehungsweise waren im industriellen Einsatz. Es hat sich außerdem herausgestellt, dass Entwurfsmuster und insbesondere *Gamma-Muster* in fast allen heutzutage existierenden Softwaresystemen vorhanden sind, obwohl sie bei der Entwicklung der Systeme nicht explizit verwendet worden sind. Zusätzlich zur weiten Verbreitung von Entwurfsmustern kann eine Entwurfsmustererkennung für unterschiedliche Reverse-Engineeringaufgaben eingesetzt werden.

Zurückgewonnene Entwurfsdokumente, die mit Entwurfsmustern angereichert worden sind, dienen zum besseren Verständnis des zu analysierenden Softwaresystems. Dieser Vorteil kommt vor allem bei großen Systemen zum Tragen, die von unterschiedlichen Entwicklern implementiert worden sind und keine Vorgaben bezüglich der Implementierung hatten.

Entwurfsmuster kombinieren statische und dynamische Anteile eines Softwaresystems. Beim Reverse-Engineering können Entwurfsmuster daher für die Erkennung von Systemgrenzen oder Modulen genauso verwendet werden wie zur Erkennung von Kommunikation, wie zum Beispiel Protokollen. Einige Entwurfsmuster sind zudem zentrale Stellen in Systemen, zum Beispiel das Gamma-Muster Composite, oder dienen zur Kopplung von Systemteilen (Bridge-Gamma-Muster) und können durch ihre Erkennung als Ausgangspunkt für weitere Analysen verwendet werden.

Durch eine Erkennung domänenspezifischer Entwurfsmuster sind spezielle, gezielte Analysen eines Softwaresystems möglich. Unter Berücksichtigung so genannter *Mustersprachen* [Plo, Ris00] (englisch: pattern-languages), die eine Sammlung von Mustern mit zusätzlichen Konformitätsbedingungen sind, lassen sich zum Beispiel auch Aussagen über die Qualität der Software machen.

Die Beschreibung eines Entwurfsmusters, insbesondere von Gamma-Mustern, erfolgt dabei zum größten Teil informal durch Prosa. Die informale Beschreibung erlaubt es, Entwurfsmuster flexibel für die Entwicklung eines Softwaresystems einzusetzen. Allerdings entsteht durch die hohe Flexibilität eines Softwaresystems eine hohe Anzahl so genannter *Implementierungsvarianten* im Quelltext. Implementierungsvarianten eines Entwurfsmusters werden als *Entwurfsmusterinstanzen* bezeichnet.

Reverse-Engineering eines Softwaresystems mit dem Ziel der Rückgewinnung der Entwurfsdokumente und der Erkennung darin enthaltener Entwurfsmuster erfolgt auf der Basis des Quelltextes. Eine Erkennung von Entwurfsmustern beruht also auf der Erkennung von Entwurfsmusterinstanzen in Quelltexten.

Reverse-Engineering Ansätze für die Erkennung von Entwurfsmusterinstanzen in Softwaresystemen sind meist so genannte *single-shot* Ansätze. Single-shot bezeichnet die Art der im jeweiligen Ansatz verwendeten automatischen Analyse. Der Reengineer hat während der Ausführung der automatischen Analyse auf sie keinen Einfluss. Single-shot Ansätze, die Ergebnisse in Form von Entwurfsmusterinstanzen im zu analysierenden Quelltext liefern, benötigen aufgrund der hohen Anzahl der Implementierungsvarianten

eine große Anzahl von Eingaben für eine automatische Analyse. Durch die hohe Anzahl von Eingaben skalieren die Ansätze nicht. Analysierbare Systemgrößen liegen im Bereich von 10.000 Zeilen Quelltext [Wil96, Rad99].

Eine Reduktion der Eingaben ist eine Möglichkeit, die zu analysierende Systemgröße in single-shot Ansätzen zu erhöhen. Dies führt entweder dazu, dass nicht alle Entwurfsmusterinstanzen gefunden werden oder die produzierten Ergebnisse zu unpräzise sind. Unpräzise Ergebnisse resultieren aus unpräzisen Eingaben und enthalten fälschlicherweise erkannte Entwurfsmusterinstanzen, so genannte *false-positives*. Die Präzision liegt bei derartigen Ansätzen unter 50% und vielfach sogar unter 10% [KP96, AFC98, TA99]. Dabei wird die Präzision eines Ergebnisses nicht mit angegeben, so dass der Reengineer alle Ergebnisse manuell auf ihre Korrektheit hin überprüfen muss.

Bezogen auf die Analyse großer Softwaresysteme mit mehreren hundert tausend oder millionen Zeilen Quelltext, sind die existierenden Ansätze nicht geeignet. Entweder ist die analysierbare Systemgröße zu gering oder die manuelle Überprüfung der produzierten Ergebnisse ist zu zeitaufwändig.

1.2 Ergebnisse der Arbeit

Ziel dieser Arbeit ist eine Erkennung von Entwurfsmusterinstanzen in Quelltexten großer Systeme, wobei die gefundenen Entwurfsmusterinstanzen möglichst wenig false-positives enthalten sollen. Dabei sollen Entwurfsdokumente, insbesondere Klassendiagramme, zurückgewonnen werden. Die Entwurfsdokumente werden mit Informationen über vorhandene Entwurfsmusterinstanzen angereichert und dem Reengineer bereitgestellt.

Zur Erreichung des Ziels wird ein semi-automatischer Analyseprozess vorgestellt. Der Analyseprozess basiert auf einer automatischen, inkrementellen, statischen Analyse eines Quelltextes [NSW+02]. Sie liefert frühzeitig Ergebnisse im Gegensatz zu single-shot Ansätzen [KSRP99, Wil96, SvG98], bei denen eine Analyse erst vollständig ausgeführt werden muss.

Der Reengineer ist stark in den semi-automatischen Analyseprozess eingebunden. Dadurch kann von seinen Erfahrungen während der Analyse profitiert werden. Zudem kann der Reengineer die Analyse steuern, und Hypothesen, zum Beispiel aufgrund zusätzlicher Informationen, mit in die Analyse einbringen.

Die Einbeziehung und damit Ergänzung der Ergebnisse durch zusätzliche Informationen ermöglicht es außerdem, den Analyseprozess mit anderen Analysen zu koppeln. Es können dabei die Ergebnisse für andere Analysen verwendet werden beziehungsweise die Ergebnisse anderer Analysen übernommen werden.

Die automatische Analyse des Quelltextes erfordert eine präzise Formalisierung der Entwurfsmusterinstanzen. In dieser Arbeit werden zur Formalisierung der Entwurfsmusterinstanzen Graphtransformationsregeln [Roz99] verwendet. Graphtransformationsregeln können in eine operationale Spezifikation überführt werden, wodurch ein Erkennungsal-

gorithmus für Entwurfsmusterinstanzen generierbar ist. Bei der Generierung können Optimierungen, die aufgrund der verwendeten Graphtransformationsregeln möglich sind, mit in den Erkennungsalgorithmus einfließen.

Die Anwendung einer Graphtransformationsregeln ist entweder erfolgreich oder nicht. Eine Entwurfsmusterinstanz wird also entweder gefunden oder nicht, wodurch false-positives durch den Reengineer identifiziert werden müssen. Der vorgestellte Ansatz erlaubt es, Graphtransformationsregeln einen *Genauigkeitswert* zuzuweisen. Der Genauigkeitswert ist ein Schätzwert über die relative Anzahl produzierter false-positives der Graphtransformationsregel. Die Zuordnung der Genauigkeitswerte zu Graphtransformationsregeln ist mit der Zuordnung der Fuzzywerte zu Grammatikregeln in Fuzzymengen [Zad65] vergleichbar. Die Genauigkeitswerte der Graphtransformationsregeln fließen dabei wie bei Fuzzygrammatiken die Fuzzywerte in die Präzisionswerte der produzierten Ergebnisse ein. False-positives können somit durch einen niedrigen Präzisionswert identifiziert werden.

1.3 Aufbau der Arbeit

Das folgende Kapitel 2 enthält nach einer Einordnung dieser Arbeit und einer Begriffsklärung für den Bereich des Reengineering den Stand der Forschung bei der Erkennung von Entwurfsmusterinstanzen in Quelltexten großer Systeme. Zusätzlich werden allgemeine Reverse-Engineering Ansätze auf ihre Eignung zur Erkennung von Entwurfsmusterinstanzen hin untersucht.

Kapitel 3 stellt nach einer Einführung in Entwurfsmuster den Ansatz dieser Arbeit zur Erkennung von Entwurfsmusterinstanzen in Quelltexten großer Systeme vor. Der Ansatz besteht aus einem inkrementellen Analyseprozess, in dem der Reengineer im Wechsel mit einer automatischen Analyse des Quelltextes eine Menge von Graphtransformationsregeln so anpassen kann, dass möglichst wenig false-positives produziert werden. Durch die Interaktivität kann der Reengineer zusätzliche Informationen beziehungsweise sein Wissen und seine Erfahrung mit in die Analyse einbringen.

Kapitel 4 beschreibt die Formalisierung von Entwurfsmusterinstanzen auf Basis einer Syntaxgraphrepräsentation des Quelltextes als Graphtransformationsregel. Eine Menge von Graphtransformationsregeln wird in einem Regelkatalog zusammengefasst und ist zusätzlich zum Quelltext die Eingabe für die automatische Analyse. Bei einer erfolgreichen Regelanwendung wird der Syntaxgraph des Quelltextes annotiert und die erzeugte Annotation wird damit Teil des Syntaxgraphen.

Kapitel 5 beschreibt den Kern der automatischen Analyse, den Inferenzalgorithmus. Der Inferenzalgorithmus wendet Graphtransformationsregeln für Entwurfsmusterinstanzen auf den Syntaxgraphen des Quelltextes an. Der Algorithmus arbeitet inkrementell, und kann vom Reengineer jederzeit unterbrochen werden. Nach einer Unterbrechung werden die Präzisionswerte der Ergebnisse aufgrund der Genauigkeitswerte der Graphtransfor-

mationsregeln berechnet. An einem Beispiel wird dargestellt, wie sich Änderungen des Reengineer an einzelnen Präzisionswerten bei ihrer Neuberechnung auswirken.

Die automatische Adaption der Genauigkeitswerte wird in Kapitel 6 beschrieben. Die Adaption erfolgt durch ein statistisches Verfahren basierend auf den Korrekturen des Reengineer an Präzisionswerten der Ergebnisse. Die automatische Adaption unterstützt dabei den Reengineer, da die Genauigkeitswerte für die Graphtransformationsregeln nur geschätzt werden können.

Kapitel 7 beschreibt zusätzliche Anforderungen an eine technische Realisierung des vorgestellten Ansatzes und stellt den entwickelten Prototypen vor. Neben Testergebnissen der automatischen Analyse des Prototypen wird ein allgemeiner Analyseprozess beschrieben, der sich bei einem neu zu analysierenden System zur Erkennung von Entwurfsmusterinstanzen bewährt hat. Ebenso werden Auswirkungen der automatischen Adaption der Genauigkeitswerte der Graphtransformationsregeln diskutiert.

Kapitel 8 schließt die Arbeit mit einer Zusammenfassung der Ergebnisse und offener Probleme sowie Verbesserungen und Perspektiven des Ansatzes ab.

KAPITEL 2: STAND DER FORSCHUNG

Entwurfsmuster kommen heutzutage vorwiegend bei der (Weiter-)entwicklung eines Softwaresystems zum Einsatz. Dabei sind sie das Produkt von Softwareanalysen, und sind nahezu in allen Softwaresystemen zu finden. Aufgrund der vielfältigen Einsatzmöglichkeiten von Entwurfsmustern kann eine Erkennung von Entwurfsmusterinstanzen für verschiedene Reverse-Engineering Aufgaben wie zum Beispiel die Erkennung von Systemgrenzen, Modulen oder Kommunikationen eingesetzt werden. Außerdem kann sie als Ausgangspunkt für weitere Analysen verwendet werden. Allerdings existieren bis heute nur wenige Ansätze, die sich mit der Erkennung von Entwurfsmusterinstanzen in Quelltexten beschäftigen.

Im folgenden Abschnitt erfolgt eine Einordnung dieser Arbeit in den Bereich des Reengineering. Für eine automatische Erkennung von Entwurfsmusterinstanzen ist eine Formalisierung notwendig. Entsprechende Ansätze werden auf ihre Eignung zur Erkennung von Entwurfsmusterinstanzen untersucht. Anschließend werden sowohl spezielle Ansätze, die eine Erkennung von Entwurfsmusterinstanzen in Quelltexten erlauben, vorgestellt, sowie allgemeine Ansätze auf die Möglichkeit der Entwurfsmusterinstanzerkennung in großen Softwaresystemen hin untersucht.

2.1 Reengineering

Der Softwarelebenszyklus wird mit verschiedenen Modellen beschrieben: zum Beispiel dem Wasserfallmodell [Roy70], dem Spiralmodell [Boe88] oder dem V-Modell [BD95].

Das Wasserfallmodell ist ursprünglich ein rein sequenzielles Modell ohne Zyklen, das in Weiterentwicklungen [Bal96] durch Zyklen wie im Spiralmodell ergänzt worden ist. Abbildung 2.1 zeigt ein adaptiertes, heutzutage in der Industrie häufig eingesetztes Wasserfallmodell, in dem, zusätzlich zu den “klassischen” Phasen des Modells, das Produktmanagement und die Qualitätssicherung verankert sind. Zyklen sind durch ergänzende Rückwärtsschritte, im Gegensatz zum klassischen Wasserfallmodell, zwischen den einzelnen Phasen möglich.

Das zu entwickelnde Softwaresystem wird im Wasserfallmodell auf verschiedenen Abstraktionsgraden beschrieben. Dabei bezeichnet der Begriff *Forward-Engineering* den Entwicklungsweg einer Software beginnend bei den Anforderungen über die funktionale Spezifikation und den Systementwurf hin zur Implementierung. Dabei sinkt der Abstraktionsgrad von einer informalen Anforderungsbeschreibung hin zu einer konkreten Imple-

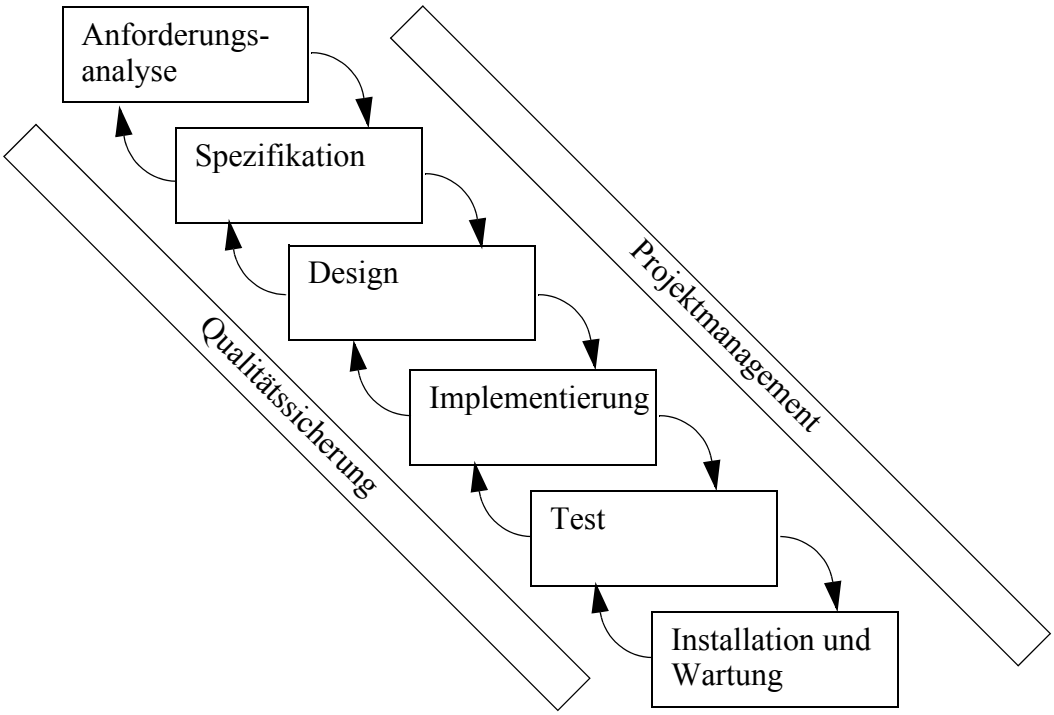


Abbildung 2.1 Erweitertes Wasserfallmodell

mentierung des Systems. In den beiden letzten Phasen, Test sowie Installation und Wartung, wird mit dem konkreten System gearbeitet.

Während der Entwicklung eines Softwaresystems werden eine Vielzahl Dokumente erzeugt. Der Begriff *Dokument* ist dabei nicht beschränkt auf Implementierungsdokumente eines Systems, sondern umfasst alle im Softwarelebenszyklus entstehenden Dokumente, angefangen bei einer informellen Anforderungsanalyse über Spezifikationen und Tests bis hin zu Wartungsdokumenten. Heutzutage wird für die Dokumentation eines Softwaresystems die *Unified Modelling Language* [UML] verwendet, die verschiedene Diagrammtypen für die einzelnen Phasen des Softwarelebenszyklus zur Verfügung stellt. Zum Beispiel werden Use-Case-Diagramme für Anforderungsdokumente verwendet, während Klassendiagramme die statische Struktur des Systems beschreiben und das Verhalten durch Sequenz-, Aktivitäts-, Kollaborations- und Statechartdiagramme spezifiziert werden kann. Die Verhaltensdiagramme können dabei auch für die Testdokumentation verwendet werden.

Im Gegensatz zum Forward-Engineering wird beim *Reverse-Engineering* ein Softwaresystem analysiert, um Repräsentationen des Systems - Dokumente - auf einem höheren Abstraktionsgrad zu erhalten.

> *Reverse engineering is the process of analysing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction.*
>
> [CC90]

Im Gegensatz zum Hardware Reverse-Engineering, aus dem der Begriff ursprünglich stammt, existieren im Software Reverse-Engineering keine standardisierten Begriffe. Die Veröffentlichung von Chikofsky und Cross [CC90] gilt als Quasistandard für die Begriffsdefinition. Chikofsky und Cross verwenden darin ein Modell mit drei Abstraktionsebenen, siehe Abbildung 2.2. Die in Abbildung 2.1 vorgestellte oberste Abstraktionsebene der Anforderungsanalyse fehlt.

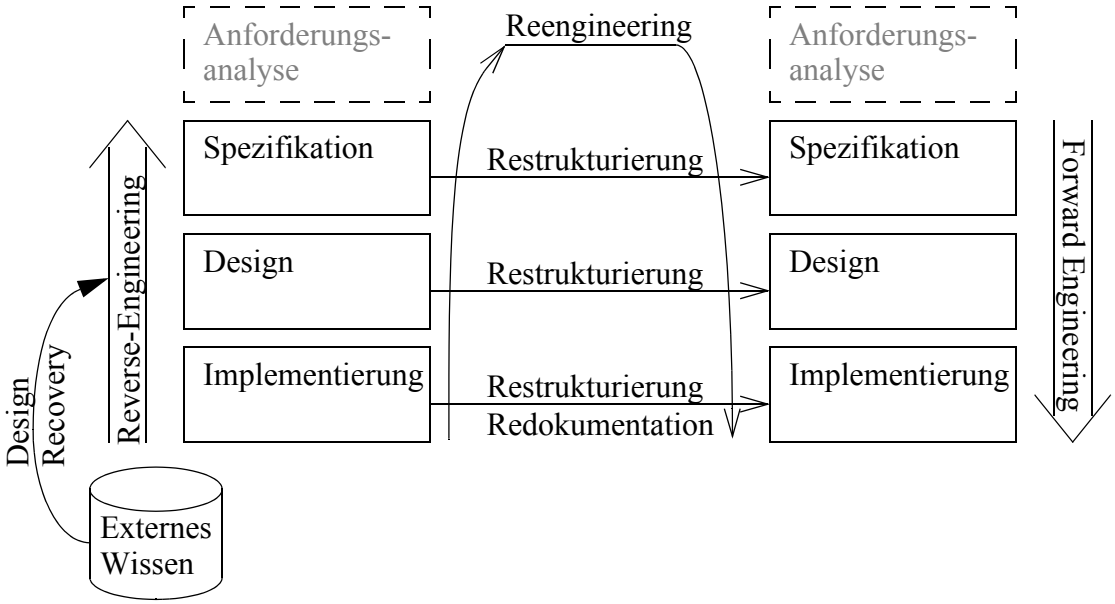


Abbildung 2.2 Begriffseinordnung nach Chikofsky und Cross

Brooks [Bro83] und Letovsky [Let86] beschreiben, dass Reverse-Engineering allgemein auf Basis von *Hypothesen* geschieht. Ein Reengineer, der ein System manuell analysiert, versucht, gezielt Hypothesen über das Verhalten eines Systems aufzustellen und sie anschließend zu festigen. Die Hypothesen entstehen dabei aufgrund von Indizien und des Vorwissens, das der Reengineer in die Analyse mit einbringt, siehe [vML99].

Redocumentation

Der Begriff *Redocumentation* (Redokumentation) bezeichnet die Erzeugung beziehungsweise Überarbeitung einer semantisch äquivalenten Darstellung eines Systems auf dem gleichen Abstraktionsniveau wie das System selbst. Auch wenn die Definition keine spe-

zifischen Angaben über die Abstraktionsebene enthält, wird die Redokumentation lediglich auf der Implementierungsebene angewendet. Beispiele für Redokumentation sind die Umgestaltung der Implementierung (Einrückungen, Umbrüche, Namenskonventionen, etc.), die Darstellung der Implementierung als Datenfluss- oder Kontrollflussdiagramm oder die Anreicherung durch Querverweise. Die einzelnen äquivalenten Darstellungen können dabei das System aus unterschiedlichen Sichten repräsentieren.

Restructuring

Die Transformation eines Systems innerhalb einer Abstraktionsebene wird als *Restructuring* (Restrukturierung) bezeichnet. Die Funktionsweise des Systems bleibt bei einer Restrukturierung unverändert. Die häufigste Restrukturierung findet auf Implementierungsebene statt, wo so genannter "Spaghetti-Code" im Sinne von Structured Design [YC79] besser modularisiert/strukturiert und damit einfacher lesbar wird. Eine andere Art der Restrukturierung ist zum Beispiel die Normalisierung von relationalen Datenbanken [EN94]. Eine Restrukturierung eines Quelltextes findet auf Implementierungsebene und eine Datenbanknormalisierung auf Designebene statt. Restrukturierung findet also auf verschiedenen Ebenen des Softwarelebenszyklus statt.

Reengineering

Unter dem Begriff *Reengineering* wird die Verbindung zwischen einer Analyse und einer anschließenden Modifikation eines Softwaresystems verstanden. Das Ziel der Analyse ist die Erstellung und/oder Überarbeitung der Repräsentation des Softwaresystems auf einer höheren Abstraktionsebene. Modifikationen werden anschließend in den abstrakteren Repräsentationen vorgenommen und fließen wiederum in die Implementierung mit ein. Die Unterschiede zur Redokumentation und zur Restrukturierung sind zum Einen die Einbeziehung mehrerer Abstraktionsebenen und zum Anderen die Änderung der Funktionalität des Softwaresystems.

Design-Recovery

Das so genannte *Design-Recovery* stellt einen Unterbereich des Reverse-Engineerings dar, bei dem Expertenwissen oder auch externes Wissen jeglicher Quellen mit in den Analyseprozess einfließt. Zusätzliche Informationen können sowohl vorhandene (externe) Dokumente wie Interviews, Telefongespräche oder auch Wissen aus dem Anwendungsbereich und persönliche Erfahrungen sein. Bei der Verwaltung und beim Einfluss des Wissens in das Reverse-Engineering werden oft Techniken aus dem Bereich "Wissensbasierte Systeme" wie *Deduktion* und *Fuzzy Reasoning* eingesetzt. Durch die Hinzunahme von externem Wissen wird zum Einen die Erstellung einer abstrakten Repräsentation eines Softwaresystems erleichtert und zum Anderen die abstrakte Beschreibung detaillierter.

Die in dieser Arbeit vorgestellte Erkennung von Entwurfsmusterinstanzen in Quelltexten großer Systeme besteht aus einem semi-automatischen Analyseprozess und ist in den Bereich des Design-Recovery einzuordnen. Im Folgenden werden Ansätze zur Formalisierung von Entwurfsmustern auf ihre Eignung zur Erkennung von Entwurfsmusterinstanzen in großen Systemen untersucht bevor Ansätze zur Entwurfsmustererkennung und allgemeine Design-Recovery Ansätze eruiert werden.

2.2 Ansätze zur Entwurfsmusterformalisierung

Im Bereich der Softwareentwicklung werden immer wiederkehrende, gleichartige Lösungen für ein Problem als *Muster* bezeichnet. Muster kommen in der Softwareentwicklung in praktisch allen Phasen des Softwarelebenszyklus vor und werden durch ihren Anwendungsbereich kategorisiert. Es existieren *Architekturmuster* [BMR+96, CKV96], *Implementierungsmuster* [Lan99] sowie *Entwurfsmuster* [GHJV95].

Muster sind in der Regel das Produkt von Analysen unterschiedlicher Softwaresysteme. Zum Beispiel sind die Entwurfsmuster aus [GHJV95] durch eine Analyse von Softwaresystemen bei IBM entstanden. Die darin beschriebenen 23 Entwurfsmuster werden als *Gamma-Muster* bezeichnet und sind eine Auswahl häufig verwendeter Entwurfsmuster. Weitere Entwurfsmuster sind in den Veröffentlichungen der Konferenzen “Pattern-Language of Program Design” zu finden [Plo].

Allgemein besteht die Beschreibung eines Musters aus 4 Teilen: einer Bezeichnung, einer Problembeschreibung, einer Lösungsbeschreibung sowie einer Beschreibung der Konsequenzen bei der Anwendung eines Musters. Die genaue Struktur und der Abstraktionsgrad einer Musterbeschreibung hängen dabei vom Anwendungsbereich des Musters ab. Im Beispiel von Gamma-Mustern umfasst die Struktur jedes Musters 13 Abschnitte, von denen die meisten informal durch Prosa beschrieben sind. Muster werden außerdem als Einheit beschrieben. Gleiche oder ähnliche Teile, die in mehreren Mustern vorkommen, werden dadurch teilweise unterschiedlich beschrieben.

Durch die informale Beschreibung von Mustern, insbesondere von Entwurfsmustern, lassen ihre konkreten Implementierungen viele Freiheitsgrade zu. Die vielen Freiheitsgrade führen dazu, dass für ein Entwurfsmuster eine sehr hohe Anzahl von *Implementierungsvarianten* im Quelltext eines Systems zu finden sind. Eine Implementierungsvariante eines Entwurfsmusters wird als *Musterinstanz* bezeichnet. Musterinstanzen unterscheiden sich zum Beispiel durch unterschiedlichen Kontrollfluss oder verwendete Bibliotheken mit unterschiedlichen Schnittstellen und Eigenschaften. Die Gründe für die Verwendung einer bestimmten Implementierungsvariante sind Performanzaspekte, Implementierungsrichtlinien oder auch der persönliche Implementierungsstil eines Entwicklers.

Zusätzlich zu Implementierungsvarianten werden bei Entwurfsmustern so genannte *Designvarianten* unterschieden. Designvarianten eines Entwurfsmusters unterscheiden sich im Design der Lösung, ohne jedoch die Lösungsidee zu verändern. Damit sind sie keine

eigenständigen Entwurfsmuster, da Muster sich in ihrer Lösungsidee für ein Problem unterscheiden müssen [Vli98] und nicht nur in einem Teilaspekt. Insbesondere Performanzaspekte können zu Designvarianten führen, die für ein konkretes Problem besser als Lösung geeignet sind. Designvarianten werden häufig in *Mustersprachen* als Alternativen vorgestellt, und ihre Abhängigkeit zu anderen Mustern diskutiert [CKV96].

Entwurfsmuster werden weitestgehend informal durch Prosa beschrieben, wodurch sie sich flexibel bei der Softwareentwicklung verwenden lassen. Dabei müssen Entwurfsmuster manuell in das Softwaresystemdesign integriert werden. Dies wird als *Instanziierung* bezeichnet. Für eine werkzeugunterstützte Instanziierung ist eine formale Beschreibung von Entwurfsmustern notwendig. Ansätze zur Formalisierung (und Instanziierung) von Mustern und insbesondere von Entwurfsmustern werden in [ACG+95] und in [Bün99, BFLS99] vorgestellt.

Zur Definition von Mustern werden in [ACG+95] *Abstract Data Objects* (ADO) verwendet. Instanzen dieser ADO's, so genannte *Abstract Data Views* (ADV) bilden dann Instanzen eines Musters in einem System. Das formale Modell, auf dem ADO's und ADV's basieren, wird in [CILS93a, CILS93b, CL95] definiert. Der Ansatz erlaubt es, ADV's durch Komposition, Mengenbildung und Vererbung zu verbinden und zu einem konkreten System zusammenzufügen. Für ADV's existieren keine adäquaten Operationen, so dass sich Musterdefinitionen nicht komponieren lassen.

In [Bün99, BFLS99] wird die *Pattern Language* (PaL) vorgestellt. PaL verwendet einen Teil der Sprache EIFFEL [Mey88] und erweitert diesen Teil um das Sprachkonstrukt "structure" (Struktur). Strukturen bestehen aus einer Menge von Klassen im objektorientierten Sinn und einer Menge von Methoden, die das dynamische Verhalten beschreiben. Strukturen können keine weiteren Strukturen enthalten, was dazu führt, dass eine Struktur einer Musterdefinition entspricht. Vererbungsbeziehungen von Strukturen werden nicht unterstützt. Ebenso existiert keine getrennte Beschreibung von Beziehungen zwischen Klassen, sondern Beziehungen werden durch Attribute in Klassen beschrieben.

Sowohl PaL wie auch ADOs und ADV's sind für die Instanziierung beziehungsweise die Verwendung von Mustern beim Softwareentwurf entwickelt worden. Durch ihre Kombinationsmöglichkeit der Musterinstanzen auf der einen Seite und der fehlenden Kombinationsmöglichkeit bei der Musterdefinition [Pre94] auf der anderen Seite sind beide Formalisierungen mit Templates vergleichbar.

In den beiden obigen Ansätzen deckt eine Musterdefinition lediglich eine Implementierungsvariante ab. Syntaktisch variierende Implementierungsvarianten eines Musters beziehungsweise äquivalente Implementierungsvarianten, in denen einzelne Anweisungen in unterschiedlicher Reihenfolge vorkommen, können mit den beiden Ansätzen nur dann erkannt werden, wenn zu jeder Variation eine eindeutige Definition existiert. Die Erkennung von Entwurfsmusterinstanzen in Quelltexten großer Systeme lässt sich daher aufgrund der hohen Anzahl Implementierungsvarianten eines Musters in der Praxis mit diesen Ansätzen nicht realisieren.

2.3 Ansätze zur Erkennung von Entwurfsmusterinstanzen

Die folgenden Ansätze zur Erkennung von Entwurfsmusterinstanzen sind *single-shot* Ansätze. Single-shot heißt, dass der Analyseprozess aus einer automatischen Analyse besteht, auf die der Reengineer während der Analyse keinen Einfluss hat. Ergebnisse zur weiteren Beurteilung oder für etwaige weitere Analysen liegen dem Reengineer erst nach einer vollständigen Analyse vor.

Das Prinzip der automatischen single-shot Analysen von Quelltexten geht auf Harandi und Ning zurück, die einen Plan-Ereignis Ansatz vorstellen [HN90]. Ereignisse besitzen Parameter wie zum Beispiel Variablenwerte, Methodennamen oder beliebige große Datenstrukturen und werden an vorgegebene Pläne geschickt, wenn ein Plan erfüllt ist. Ein Plan gilt als erfüllt, wenn die Parameter der Ereignisse, die der Plan erhält, die Bedingungen des Plans erfüllen.

Im Folgenden werden die einzelnen Ansätze zur Erkennung von Entwurfsmusterinstanzen in großen Softwaresystemen vorgestellt und ihre Vor- und Nachteile diskutiert. Die existierenden Ansätze lassen sich in drei Gruppen einteilen. Die erste Gruppe umfasst Ansätze, in denen explizit Suchalgorithmen angegeben werden müssen. Die zweite Gruppe umfasst Ansätze, die Regelausführungssysteme verwenden. Zur dritten Gruppe gehören Ansätze, die ebenfalls Regelausführungssysteme verwenden. Zusätzlich verwenden sie Heuristiken, um die Regelkomplexität und/oder den Graphen des zu analysierenden Softwaresystems und damit die Laufzeit der Regelausführung und deren Wartung zu reduzieren.

2.3.1 Skript basierte Ausführung

Das SPOOL Projekt an der Universität von Montreal, Kanada enthält einen Ansatz zur Erkennung von Entwurfsmusterinstanzen in Quelltexten, der statische Strukturen sowie das dynamische Verhalten von Entwurfsmustern mit berücksichtigt [KSRP99]. Der Quelltext wird mit Hilfe eines Parsers[1] in ein UML Klassendiagramm [UML] überführt. Die UML Klassendiagramme werden in einer ASCII Repräsentation als erweitertes UML/CDIF Modell [EIA] abgelegt. Das Klassendiagramm besteht aus Klassen mit Attributen und Methoden. Zeiger auf andere Klassen werden als Referenzbeziehungen dargestellt. Eine Abstraktion des Klassendiagramms findet nicht statt. Solche Klassendiagramme werden als *rudimentär* bezeichnet.

Entwurfsmuster werden bei diesem Ansatz als so genannte Design Komponente (englisch Design-Component) [KS98] in einem Repository abgelegt. Design Komponenten können Teile von (Teil)Klassendiagrammen, Suchabfragen und auch Veränderungsoperationen auf dem Modell beinhalten.

1. Zur Zeit existiert ein Parser für C++ Quelltexte.

Um Design Komponenten im UML/CDIF Modell des Quellcodes zu finden, existieren für den Anteil der Klassendiagramme vordefinierte Algorithmen, die in der Skriptsprache Perl [WSC96] implementiert sind. Diese vordefinierten Algorithmen entsprechen im Wesentlichen denen, die auch in Regelsystemen verwendet werden.

Zusätzliche Anteile einer Design Komponente, wie zum Beispiel weitere Suchanfragen und Analysen, werden ebenfalls durch Skripte implementiert. Da Klassendiagramme keine dynamischen Anteile enthalten, müssen diese durch zusätzliche Analyseskripte implementiert werden.

Skripte einer Design Komponente sind keine kapselnden Einheiten, so dass Skripte einer Design Komponente auf Skripte anderer Design Komponenten zugreifen können. Dies erlaubt eine Komposition von Mustern, wobei die Kompositionsabhängigkeiten der Muster beziehungsweise Skripte nicht explizit abgelegt werden können.

Durch die Verwendung regulärer Ausdrücke in einem Perl-Skript können gering differierende Implementierungen eines Entwurfsmusters zusammengefasst werden. Für stark differierende Implementierungsvarianten muss jeweils ein eigenes Perl-Skript implementiert werden.

Die gefundenen Entwurfsmusterinstanzen können mit Hilfe des RIGI-Systems [WKJ+95] dargestellt werden. False-positives treten bei diesem Ansatz insbesondere durch die Verwendung regulärer Ausdrücke bei der Analyse des dynamischen Verhaltens auf.

Vergleichbar mit dem Ansatz in [KSRP99] ist der Ansatz in [HHL02, HHHL03], bei dem Entwurfsmusterinstanzen als Algorithmen formuliert werden. Der Ansatz kombiniert statische und dynamische Analysen, wobei die statische Analyse im Wesentlichen für die Erkennung von Entwurfmusterinstanzkandidaten dient, die durch eine sich anschließende dynamische Analyse validiert werden.

Sowohl die Analyse statischer als auch dynamische Anteile eines Entwurfsmusters erfolgt durch Algorithmen, die auf Basis eines abstrakten Syntaxgraphen (statisch) und auf konkreten Aufrufgraphen (dynamisch), die während der dynamischen Analyse erzeugt werden, formuliert werden. Die Algorithmen für einzelne Entwurfsmusterinstanzen werden vom Reengineer geschrieben und sind den Skripten aus [KSRP99] ähnlich. Eine Definition von Teilmustern erlaubt der Ansatz nicht, was zu mehrfacher Definition und Erkennung von Teilmustern führt.

Der größte Nachteil der obigen Ansätze ist die Wartbarkeit der verwendeten Skripte beziehungsweise Algorithmen, da die Abhängigkeiten zwischen Skripten nicht explizit angegeben werden können. Eine einfache Wiederverwendung einzelner Skripte für die Analyse weiterer Systeme ist daher schwierig. Werden diese Skripte für ein spezielles System optimiert, ist eine Wiederverwendung oder auch Anpassung an ein anderes System nur mit erheblichem Aufwand möglich, was einem Reverse-Engineering der Skripte entspricht.

2.3.2 Regelausführungssysteme

Bei der Verwendung von Regelausführungssystemen werden Entwurfsmusterinstanzen durch Regeln definiert, und das Ausführungssystem enthält Algorithmen, um die Regeln anzuwenden.

Paul und Prakash verwenden für die Erkennung von Mustern in Quelltexten das System SCRUPLE [PP94]. Musterinstanzen werden in einer textuellen Sprache definiert, die, ähnlich wie bei Regulären Ausdrücken, Platzhalter enthalten können. Im Gegensatz zu Regulären Ausdrücken existieren Platzhalter nicht nur für Buchstaben- und Zahlenketten, sondern die verwendeten Platzhalter können auch syntaktische Einheiten wie Datentypen, Deklarationen, Variablen oder Funktionen enthalten.

Das SCRUPLE System verwendet einen nicht-deterministischen, endlichen Automaten, den so genannten Code Pattern Automaton (CPA), zur Mustersuche. Der Nichtdeterminismus des Automaten wird dazu verwendet, die Platzhalter richtig zu belegen. Bei der technischen Realisierung wird für den Nichtdeterminismus ein backtracking Algorithmus verwendet.

Bei einer Analyse mit Hilfe des SCRUPLE Systems wird der Quelltext in einen abstrakten Syntaxgraphen (ASG) durch einen Parser abgebildet, und auch die Musterdefinition wird mit Hilfe eines Parsers in einen CPA abgebildet. Anschließend wird der Automat von einem Interpreter ausgeführt. Das Ergebnis ist eine Menge von Quelltextausschnitten, die manuell vom Reengineer auf false-positives hin untersucht werden.

Der beschriebene Ansatz hat aufgrund der verwendeten syntaktischen Einheiten als Platzhalter eine polynomielle Laufzeit in Bezug auf die Anzahl der Knoten des abstrakten Syntaxgraphen. Durch die starke Orientierung an der Syntax lassen sich gering differierende Implementierungsvarianten wie unterschiedliche Reihenfolgen in Anweisungen nur erkennen, wenn jede Variante als eigenständiges Muster definiert wird. Komposition von Musterinstanzen wird nicht unterstützt. Dass heißt, dass die Sprache nur eine Definition vollständiger Musterinstanzen unterstützt. Dies führt bei der Mustersuche dazu, dass gemeinsame Teilmuster mehrfach gesucht werden müssen. Für die Erkennung von Entwurfsmusterinstanzen in Quelltexten großer Systeme ist dieser Ansatz daher nicht geeignet.

Kremer und Prechelt stellen in [KP96] einen Ansatz zur Erkennung struktureller Entwurfsmuster in objektorientierten Softwaresystemen basierend auf PROLOG (Programming in Logic) [WC94] vor. Ein Parser extrahiert Klassen, Attribute und Methoden und legt die Informationen als PROLOG Fakten ab. Musterinstanzen werden als PROLOG Regeln definiert, und der Ausführungsmechanismus wendet die Regeln auf die Fakten an.

Die Genauigkeit bei der Erkennung der strukturellen Entwurfsmuster in C++-Quelltexten liegt zwischen 14% - 50%. Dass heißt, dass es sich bei einer gefundenen Entwurfsmusterinstanz in mehr als der Hälfte der Fälle um ein false-positive handelt. Dadurch, dass keine dynamischen Anteile eines Musters definiert werden können, ist die Anzahl zu extrahie-

render Fakten und die Anzahl der Regeln klein, und es lassen sich große Systeme analysieren. Durch die Berücksichtigung dynamischer Anteile wächst die Anzahl der Fakten und damit die Anzahl der Regeln. Durch den allgemeinen Ausführungsmechanismus von PROLOG, der auf einem Backtrackingverfahren beruht, ist der Ansatz dann nur noch für kleine Systeme anwendbar.

Graphersetzungssysteme

Bei der Verwendung von Graphrepräsentationen des Quelltextes werden oft Graphersetzungssysteme [Roz99] zur automatischen Analyse verwendet. Graphersetzungssysteme wenden Graphtransformationsregeln auf einen Graphen an.

Die Ausführung einer Graphtransformationsregel ist nicht-deterministisch. Graphersetzungssysteme verwenden daher ein Backtracking-Verfahren, um Graphtransformationen auszuführen. Ein solches System ist das Graphersetzungssystem PROGRES [SWZ95], dass an der RWTH Aachen, Deutschland entwickelt worden ist. Alternativ kann die Ausführung auch als Constraint Satisfaction Problem (CSP) formuliert werden, wie im Falle des AGG Systems [AGG, Tän00, Rud97] der TU Berlin, Deutschland.

Die Anwendung einer Graphtransformationsregel geschieht in zwei Phasen. Erst wird eine Anwendungsstelle der Regel im Graphen gesucht, und anschließend werden die Veränderungen, die in der Regel angegeben sind, ausgeführt. Das Suchen der Anwendungsstelle im Graphen entspricht dem NP-vollständigen Problem der Teilgraphensuche [Meh84]. Durch die NP-Vollständigkeit sind Graphersetzungssysteme ohne optimierte Suche nicht für eine Erkennung von Entwurfsmusterinstanzen in großen Systemen anwendbar.

Das Graphersetzungssystem PROGRES optimiert die Teilgraphensuche, indem es als Kontext für eine Regelanwendung eine Menge von Knoten verwendet. Diese Knoten markieren einen Teil der Anwendungsstelle. Bei der Regelausführung wird ein Teilgraph - ausgehend von den markierten Knoten - gesucht, der dann die Anwendungsstelle der Regel ist.

Durch die Möglichkeit, zusätzlich zur Angabe eines Kontextes, einen expliziten Kontrollfluss der Regeln anzugeben, ist die Laufzeit für die Suche der Anwendungsstelle im durchschnittlichen Fall polynomiell. Weitere Details zur optimierten Teilgraphensuche in PROGRES sind in [Zün96] zu finden. Die Angabe eines Kontextes für die Regelanwendung stellt zwar eine Einschränkung bezüglich der Graphgrammatiktheorie dar, in der Praxis hat sich allerdings gezeigt, dass ein Kontext immer angegeben werden kann.

Radermacher [Rad99] verwendet das Graphersetzungssystem PROGRES als Regelausführungssystem für die Erkennung von Entwurfsmusterinstanzen. Der Quelltext wird in eine interne Graphstruktur geparst, die an das Meta-Modell von UML angelehnt ist und einem abstrakten Syntaxgraphen entspricht. Musterinstanzen werden als Graphtransformationsregeln definiert, wobei die Regeln keine Modifikationen enthalten dürfen. Die ge-

fundenen Anwendungsstellen werden nach erfolgreicher Suche durch spezielle Knoten markiert - so genannte *Annotationen*. Durch die Möglichkeit, Annotationen in Regeln zu verwenden, ist es möglich, Musterinstanzen aus Teilmustern zu komponieren, wodurch eine Abhängigkeit der Regeln entsteht. Diese Abhängigkeit definiert gleichzeitig eine optimierte Ausführungsreihenfolge der Regeln.

Der vorgestellte Ansatz erlaubt es, statische und dynamische Anteile eines Entwurfsmusters, durch Regeln zu definieren. Für große Systeme ist der Ansatz jedoch nicht anwendbar, weil PROGRES relevante Ergebnisse erst nach einer vollständigen Analyse liefert und der Ansatz damit ein single-shot Ansatz ist. Außerdem verwendet PROGRES die objektorientierte Datenbank GRAS [KSW96] zur Speicherung von Graphen verwendet. Durch die Datenbank ist der Zugriff auf Knoten zeitaufwändig, was bei einer großen Anzahl von Regeln zu sehr langen Laufzeiten führt.

Ein weiterer Nachteil des Ansatzes ist, das Regeln, die präzise ein dynamisches Verhalten beschreiben, sind aufgrund der Verwendung eines abstrakten Syntaxgraphen oft sehr groß, was zu einem Problem bei einer etwaigen Fehlersuche und Wartung der Regeln führt. Durch adäquate kleinere Regeln werden die Ergebnisse der Analyse unpräziser; es werden mehr false-positives produziert. Angaben über die Präzision von Ergebnissen liefert der Ansatz nicht.

2.3.3 Verwendung von Heuristiken

Bei der Verwendung abstrakter Syntaxgraphen zur Repräsentation von Quelltexten tritt das Problem auf, dass der abstrakte Syntaxgraph eines Quelltexts sehr komplex werden kann. Insbesondere bei Regeln, die Methodenrümpfe für dynamische Anteile von Mustern enthalten, können diese sehr komplex und unübersichtlich werden. Werden solche komplexen Regeln weggelassen, verkürzt dies im Allgemeinen die Zeit für eine Analyse beziehungsweise es steigt die analysierbare Systemgröße. Der Nachteil wie zum Beispiel in [KP96] ist, dass es dann entweder gar nicht möglich ist, Musterinstanzen zu erkennen, die sich lediglich dynamisch unterscheiden, oder die Ergebnisse sehr unpräzise sind.

Ein Kompromiss zwischen der Verwendung komplexer und damit präziser Regeln für die Definition dynamischer Anteile von Mustern und der Nichtberücksichtigung dynamischer Anteile ist es, Heuristiken oder Metriken zu verwenden. Insbesondere sind sie schnell zu berechnen und verkürzen damit die Laufzeit der Analyse. Allerdings werden die Analyseergebnisse unpräziser, da Heuristiken weniger Informationen berücksichtigen.

Bei Seemann und von Gudenberg [SvG98] werden anstatt eines abstrakten Syntaxgraphen Aufruf- und Erzeugungsgraphen für die Analyse dynamischer Anteile eingesetzt. Aufgrund von Methodenaufrufen, Objekterzeugungen und strukturellen Beziehungen zwischen Klassen werden Teilmuster, aber auch Verfeinerungen von Beziehungen zwischen Klassen, definiert. Die Informationen über Methodenaufrufe, Objekterzeugungen und strukturelle Beziehungen werden durch einen Parser aus dem Quelltext extrahiert und

in einer internen Graphstruktur abgelegt. Definitionen für Entwurfsmusterinstanzen bauen auf Teilmustern auf.

Durch die geringe Anzahl daraus abgeleiteter Regeln für ein Graphersetzungssystem zur Erkennung von Musterinstanzen in Quelltexten ist der Ansatz für größere Systeme geeignet. Die Ergebnisse der Analyse müssen allerdings manuell vom Reengineer auf Korrektheit überprüft werden und enthalten keine Aussagen über ihre Präzision.

Im Ansatz von Antoniol et al. werden dynamische Informationen durch Metriken ausgedrückt [AFC98, TA99]. Bei der Analyse großer C++-Bibliotheken wurde eine Genauigkeit von durchschnittlich 55% erreicht. Dies ist eine Verbesserung der Ergebnisse von Krämer und Prechelt [KP96], die durchschnittlich eine Genauigkeit von ca. 23% erreichen. Da aber keine Aussage über die Präzision einzelner Ergebnisse gemacht werden kann, muss der Reengineer alle Ergebnisse manuell überprüfen.

2.4 Allgemeine Design-Recovery Ansätze

Im Bereich des Design-Recovery beziehungsweise des Programmverstehens existiert eine Vielzahl von Ansätzen. Bei den verschiedenen Ansätzen werden unterschiedliche Techniken wie zum Beispiel Clustering [SL77], Slicing [Ber95] oder Pattern Matching [Tho68] verwendet. Teilweise werden Metriken verwendet, um schnell quantifizierte Aussagen über die zu analysierende Software zu machen. Ein Überblick über die verschiedenen, aktuell gebräuchlichen Ansätze und Grundlagen ist in [Kos00] zu finden. Der Überblick ist im Wesentlichen eine Zusammenfassung der Veröffentlichungen des International Workshop on Program Comprehension (IWPC) und der Working Conference on Reverse-Engineering (WCRE).

Speziell auf einzelne Softwaresysteme abgestimmte Ansätze [DFP+02, IG02] sind schwer beziehungsweise gar nicht auf andere Systeme zu übertragen. Daher werden im Folgenden allgemeine Ansätze und Projekte aus dem Bereich des Design-Recovery und Programmverstehens vorgestellt und auf die Möglichkeit der Erkennung von Entwurfsmusterinstanzen in Quelltexten hin untersucht.

Die vorgestellten Ansätze und Projekte beruhen dabei auf einer Graphrepräsentation des Quelltextes. Andere Ansätze, die zum Beispiel auf regulären Ausdrücken basieren, liefern aufgrund der Notwendigkeit der Analyse dynamischer Abläufe für die Erkennung von Entwurfsmusterinstanzen zu unpräzise Ergebnisse beziehungsweise eignen sich nicht für eine Analyse großer Systeme.

2.4.1 GRASPR Projekt

Ziel des GRASPR Projekts der Universität Waterloo, Kanada ist die Erkennung von häufig verwendeten Algorithmen in Quelltexten (englisch: common-computation-structures), wie zum Beispiel Such- oder Sortieralgorithmen [Wil92, Wil96].

Da in Algorithmen sowohl der Kontrollfluss als auch der Datenfluss die Grundlage bei der Erkennung sind, werden im GRASP Projekt attributierte Flussgraphen [Wil92] verwendet. Dabei kann entweder der Kontrollfluss als Graph und der Datenfluss durch Attribute, oder der Datenfluss als Graph und der Kontrollfluss durch Attribute repräsentiert werden. Wills [Wil96] verwendet die erste Variante und repräsentiert das zu analysierende System als attributierten Flussgraphen. Die zu erkennenden Algorithmen werden als attributierte Flussgraphgrammatiken definiert. Jede Grammatik repräsentiert einen Algorithmus und besteht aus einer Menge von Regeln. Ein Regelausführungsmechanismus wendet die Regeln auf den attributierten Flussgraphen des Programms an.

Im Allgemeinen liegt dabei jedem Regelausführungssystem das NP-vollständige Problem der Teilgraphensuche zugrunde [Meh84]. Aufgrund der NP-Vollständigkeit und der teilweise komplexen Regeln ist der Ansatz nur für Systemgrößen von 10.000 Zeilen Quelltext anwendbar. Bei einer solchen Systemgröße sind die Ergebnisse einer abgeschlossenen Analyse allerdings sehr präzise, was auf die gute Abstraktionseigenschaft der attributierten Flussgraphen zurückzuführen ist.

Bei der Analyse handelt es sich um einen single-shot Ansatz, bei der die Ergebnisse dem Reengineer erst nach Ablauf der Analyse präsentiert werden. Für eine Erkennung von Entwurfsmusterinstanzen wäre dieser Ansatz aufgrund der hohen Präzision der Ergebnisse und der Kombination von Kontroll- und Datenfluss gut geeignet. Allerdings ist die Einschränkung der Systemgröße mit dem Ziel der Analyse großer Systeme nicht erfüllbar. Außerdem sind die verwendeten Flussgraphgrammatiken anfällig gegenüber Änderungen. Schon kleine Änderungen können zu einem Fehlschlagen der Analyse führen.

2.4.2 Bauhaus Projekt

Das Bauhaus Projekt [GK97], dass an der Universität Stuttgart, Deutschland iniziiert worden ist, befasst sich mit der Rückgewinnung von Architekturinformationen aus Quelltexten. Im Projekt werden Beschreibungsmittel, Analysetechniken und dazugehörige Werkzeuge entwickelt. In einem semi-automatischen Prozess werden Sichten (englisch: views) auf Legacy-Systeme hergeleitet und Komponenten identifiziert. Die gewonnenen Informationen dienen in erster Linie dazu, etwaige Seiteneffekte bei Änderungen abzuschätzen, wobei alle Informationen persistent gespeichert werden. Als Basis für weitergehende Analysen werden Techniken aus dem Übersetzerbau wie Kontroll- und Datenflussanalysen eingesetzt.

Zur Zeit existieren Ansätze zur Erkennung abstrakter Datentypen beziehungsweise abstrakter Datenobjekte [CEK+00]. Ein weiterer Schwerpunkt des Projekts bildet die Identifizierung von Teilsystemen zur Reduktion der Systemgröße bei weiterführenden Analysen. Ebenso existieren Bemühungen, Redokumentation und Restrukturierung zu unterstützen.

Obwohl sich das Bauhaus Projekt als Schwerpunkt mit der Rückgewinnung der Architektur eines Softwaresystems beschäftigt, existiert zur Zeit keine Veröffentlichung, die eine

Erkennung von Architekturmusterinstanzen oder Entwurfsmusterinstanzen in Quelltexten thematisiert. Aufgrund der verwendeten Techniken des Übersetzerbaus in den Basisanalysen und bereits existierender Werkzeuge wäre eine Erkennung von Entwurfsmusterinstanzen vergleichbar mit dem Ansatz von Wills [Wil96], siehe 2.4.1.

2.4.3 RIGI Projekt

Das RIGI Projekt [MK88, Won98, WKJ+95], das an der Universität Victoria, Kanada entwickelt worden ist, basiert auf dem interaktiven RIGI-System, das unter anderem zum Reverse-Engineering von Softwaresystemen angewendet werden kann.

Das RIGI-System erlaubt die graphische Darstellung von Artefakten und eine interaktive Navigation durch die Darstellung. Es kann dadurch als universeller Graph-Browser, bei dem sich die Darstellung einzelner Knoten und Kanten sowie deren Anordnung beliebig anpassen lässt, angesehen werden. Das System beinhaltet eine Menge vordefinierter Standarddarstellungen und Anordnungen, so dass es sich auch für einen prototypischen Einsatz eignet.

Bezogen auf das Reverse-Engineering von Softwaresystemen kann es sich bei den darzustellenden Artefakten zum Beispiel um einzelne Softwarekomponenten, Teilsysteme, Klassen, Attribute, Prozeduren, Schnittstellen oder Protokolle handeln. Die Granularität der Artefakte ist nicht festgelegt. Beziehungen zwischen den Artefakten können zum Beispiel Kontroll- und Datenabhängigkeiten, aber auch Nachrichtenwege sein.

Ziel des RIGI Projekts ist es, Artefakte und ihre Beziehungen zu identifizieren und in geeigneter Form im RIGI System zu präsentieren. Dies ist die erste Phase einer Analyse. In einer zweiten Phase werden die meist sehr komplexen und unübersichtlichen Darstellungen manuell oder semi-automatisch zusammengefasst, um sie in einer geeigneten Art und Weise präsentieren zu können. Die Ergebisse können dann vom Reengineer interpretiert werden. Die erste Phase ist dabei von einer konkreten Programmiersprache abhängig, wobei ein Parser die Artefakte extrahiert und in einer Datenbank ablegt. In der zweiten Phase kommen sprachunabhängige Kompositionsalgorithmen zum Einsatz.

Das RIGI-System ist bereits in mehreren Reverse-Engineering Ansätzen zum Einsatz gekommen [MOTU93, TWSM94, KMS02], allerdings existiert zur Zeit kein Ansatz zur Erkennung von Entwurfsmusterinstanzen. Das RIGI-System hat dabei den Nachteil, dass das Parsen in der ersten Phase der Analyse erfolgt und somit das gesamte System in der Datenbank abgelegt werden muss. Bei großen Softwaresystemen ist ein solches Vorgehen - aufgrund der großen Datenmengen - unpraktikabel. Desweiteren gehen die Beziehungen der Artefakte zum Quelltext verloren, wodurch eine Rückverfolgung der Analyseergebnisse zum Quelltext nicht möglich ist.

2.4.4 GUPRO Projekt

Ähnlich wie beim RIGI Projekt wird beim GUPRO Projekt [EGW96], iniziiert an der Universität Koblenz-Landau, Deutschland der Quelltext durch Parser in eine Graphrepräsentation überführt. Einzelne Parser können aus der Sprache PDL [Dah95] generiert werden. Werkzeuge, die Analysen durchführen, arbeiten auf der Graphrepräsentation, wobei die Anfragesprache GReQL [EKW99] zur Verfügung steht. Das GUPRO Projekt umfasst dabei Werkzeuge, die allgemein dem Programmverstehen dienen und nicht auf den Bereich des Design-Recovery beschränkt sind.

GUPRO ist durch die Verwendung der Parsergenerierungssprache PDL mit geringem Aufwand an verschiedene Programmiersprachen anpassbar, allerdings fehlt die Möglichkeit der visuellen Darstellung der Graphrepräsentation, was für einen interaktiven Ansatz nachteilig ist. Informationen können nur über die Anfragesprache extrahiert werden und komplexe textuelle Anfragen sind wartungsaufwändig. Bezogen auf die Erkennung von Entwurfsmusterinstanzen existieren die gleichen Nachteile wie beim RIGI System.

Ein weiterer Aspekt, der in diesem Projekt behandelt wird, ist die Kopplung auf Graphen basierender Analysen. Dazu wurde die Graph Exchange Language (GXL) zum Austausch von Graphen entwickelt [Win01, WKR01]. GXL basiert auf XML und soll es ermöglichen, unterschiedliche Werkzeuge, die auf Graphen arbeiten, zu verbinden, um bessere Analyseergebnisse zu erreichen.

2.4.5 VARLET Projekt

Ein Ansatz, der den Reengineer in den Analyseprozess integriert und damit kein single-shot Ansatz ist, ist im VARLET Projekt entwickelt worden. Das VARLET Projekt [Jah99] an der Universität Paderborn, Deutschland und an der Universität Victoria, Kanada beschäftigt sich mit dem Datenbank-Reengineering. VARLET ist bereits erfolgreich bei der Analyse kommerzieller Datenbanksysteme eingesetzt worden und war die Ausgangsbasis für die Entwicklung des in dieser Arbeit vorgestellten Ansatzes zur Erkennung von Entwurfsmusterinstanzen großer Softwaresysteme.

Ebenso wie Legacy-Softwaresysteme existieren Legacy-Datenbanksysteme, zu denen unvollständige beziehungsweise keine Dokumentationen existieren. Ziel des Datenbank Reverse-Engineering ist es, ein abstraktes Modell des Datenbankschemas in Form eines Entity-Relationship Diagramms oder eines Klassendiagramms zu erstellen. Als Informationsquelle steht die Datenbank mit ihren Relationen und Daten sowie Anfragen und Konsistenzbedingungen zur Verfügung. Letztere werden in der Sprache SQL formuliert [BED94] und werden als Programmcode der Datenbank bezeichnet.

Bei der Analyse von Datenbanksystemen besteht häufig das Problem, dass sich verschiedene Informationen wiedersprechen oder unsicher sind. Im VARLET Projekt wird für den Umgang mit solchen Informationen possibilistische Logik eingesetzt [Zad78]. Durch so genannte Generic Fuzzy Reasoning Nets (GFRN) kann der Datenbank-Reengineer die Verarbeitung der Informationen spezifizieren.

Informationen, insbesondere Informationen über Beziehungen von Relationen, werden durch die Analyse des Datenbank Programmcodes extrahiert. So genannte *Clichés* werden im Programmcode durch das Graphersetzungssystem PROGRES [SWZ95, Zün96] extrahiert. Der Programmcode wird in einem abstrakten Syntaxgraphen abgebildet und Regeln spezifizieren einzelne Clichés. Es werden dabei keine wesentlichen Optimierungen in der Suche vorgenommen, weil der zu analysierende Programmcode jeweils nur aus weniger als hundert Zeilen besteht.

Der Ansatz ist für die Erkennung möglicher Speicherprobleme in Implementierungen für Java-Cards angewendet worden, siehe [JNW00]. Das verwandte GFRN ist dabei sehr klein und enthält lediglich einfache Schlussfolgerungen, aber eine große Menge von Clichés, um möglichst viele Implementierungsvarianten zu erkennen. Die möglichen Speicherlecks sind als Implementierungsmuster spezifiziert worden, und somit ist der Ansatz mit der Erkennung von Entwurfsmusterinstanzen in Quelltexten vergleichbar.

Es hat sich dabei herausgestellt, dass der Ansatz nur für kleine Programmgrößen erfolgreich ist. Dies liegt im Wesentlichen an der hohen Anzahl zu erkennender Clichés, die vor der Auswertung des GFRN aus dem Quelltext extrahiert werden müssen. Diese vorab stattfindende Extrahierung lässt den Ansatz bei seiner Verwendung für die Erkennung von Entwurfsmusterinstanzen quasi zu einem single-shot Ansatz werden. Das Problem ist Ausgangspunkt für den in dieser Arbeit beschriebenen Ansatz zur Erkennung von Entwurfsmusterinstanzen in Quelltexten großer Systeme gewesen.

2.5 Zusammenfassung

Bei der Erkennung von Entwurfsmusterinstanzen in Quelltexten großer Softwaresysteme treten mehrere Probleme auf. Die Beschreibung eines Entwurfsmusters ist weitgehend informal. Dadurch sind sie flexibel einsetzbar, aber es existieren zu einem Entwurfsmuster eine Vielzahl von Design- und Implementierungsvarianten in Softwaresystemen. Hinzu kommt, dass einige Entwurfsmuster sich nur aufgrund ihres dynamischen Verhaltens unterscheiden, was ihre eindeutige Erkennung in Quelltexten erschwert, vor allem wenn sie nur informal beschrieben sind.

Ansätze, die sich direkt mit der Erkennung von Entwurfsmusterinstanzen in Quelltexten befassen und präzise Ergebnisse liefern, skalieren nicht. Ansätze, die Heuristiken verwenden, skalieren besser, liefern aber unpräzise Ergebnisse in Form von false-positives. Außerdem sind die verwendeten Heuristiken, wie zum Beispiel Namenskonventionen, strukturelle Eigenschaften und Aufrufgraphen fest in die Systeme implementiert und lassen sich nicht einfach durch den Reengineer ändern beziehungsweise an ein konkretes Reverse-Engineering Projekt anpassen.

Allgemeine Ansätze zum Design-Recovery und Programmverstehen enthalten keine explizite Unterstützung zur Erkennung von Entwurfsmusterinstanzen in Quelltexten und bieten keine Unterstützung für eine adäquate Formalisierung beziehungsweise lassen sich nicht für die Analyse großer Systeme anwenden.

KAPITEL 3: INKREMENTELLE ANALYSE

Das vorherige Kapitel hat gezeigt, dass kein Ansatz existiert, der eine Erkennung von Entwurfsmusterinstanzen in großen Systemen adäquat unterstützt. Die existierenden Ansätze sind single-shot Ansätze und liefern entweder sehr präzise Ergebnisse, skalieren aber nicht, oder sie verwenden Heuristiken, wodurch sie besser skalieren, aber unpräzise Ergebnisse produzieren.

Am Beispiel der Gamma-Muster wird die Idee der in dieser Arbeit vorgestellten Erkennung von Entwurfsmusterinstanzen in Quelltexten großer Systeme erläutert. Der Ansatz besteht aus einem inkrementellen Analyseprozess. Der Reengineer ist in den semi-automatischen Analyseprozess stark eingebunden, wodurch er Hypothesen mit in den Analyseprozess einbringen und die automatische Analyse steuern kann.

3.1 Gamma-Muster

Gamma-Muster [GHJV95] sind in drei Teilbereiche eingeteilt: erzeugende Muster, strukturelle Muster und Verhaltensmuster. Jedes Gamma-Muster besteht aus 13 Beschreibungsteilen: Name des Musters und seine Klassifizierung, Zweck, weitere bekannte Namen, Motivation, Anwendbarkeit, Struktur, beteiligte Teile, Zusammenwirken der beteiligten Teile, Konsequenzen bei der Anwendung, Implementierungstipps, Beispiel Quelltext, bekannte Verwendungen, verwandte Muster.

Die Beschreibungsteile selbst besitzen einen unterschiedlichen Detaillierungsgrad, der von Gamma-Muster zu Gamma-Muster variiert. Beispielimplementierungen werden in C++ angegeben. Bei der Struktur wird in der Regel ein OMT-Diagramm verwendet. Neuere Literatur verwendet UML-Klassendiagramme anstatt OMT-Diagramme. Im Gegensatz zur einheitlichen Verwendung von OMT-Diagrammen zur Beschreibung der Struktur eines Gamma-Musters werden bei der Beschreibung des Zusammenwirkens beteiligter Teile teilweise Kollaborationsdiagramme, teilweise Prosa und teilweise auch beides verwendet. Damit ist die Beschreibung eines Gamma-Musters weitgehend informal.

Gamma-Muster sind in einem Katalog nach zwei Kriterien klassifiziert, siehe Abbildung 3.1. Auf der einen Seite werden Gamma-Muster anhand ihres Zwecks in drei weitere Untergruppen, die Erzeugenden Muster, die Strukturellen Muster und die Verhaltensmuster, geteilt. Auf der anderen Seite werden die Gamma-Muster anhand ihres Anwendungsbereichs eingeteilt in Muster, die sich mit Beziehungen auf Klassenebene, und Muster, die sich mit Beziehungen auf Objektebene befassen. Muster, die ihre Beziehungen durch Vererbung etablieren, werden der ersten Gruppe zugeordnet. Die Beziehungen

stehen dabei schon zur Übersetzungszeit fest und lassen sich nicht mehr ändern. Bei Mustern der zweiten Gruppe werden die Beziehungen in der Regel zur Laufzeit geändert.

		Zweck		
		Erzeugung	Struktur	Verhalten
Anwendungsbereich	Klasse	Factory Method	Adapter (Klasse)	Interpreter Template Method
	Objekt	Abstract Factory Builder Prototype Singleton	Adapter (Objekt) Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Abbildung 3.1 Klassifizierung von Gamma-Mustern

Die 23 Entwurfsmuster aus [GHJV95] sind informal durch Prosa beschrieben und lassen sich dadurch flexibel auf die jeweilige Situation und das Umfeld, in dem sie eingesetzt werden sollen, anpassen. So existieren zu einem Entwurfsmuster mehrere Designvarianten. Zum Beispiel sind in der Strukturbeschreibung des Adapter-Musters zwei Designvarianten beschrieben. Gemäß Abbildung 3.1 gehört eine Variante zur Klasse der Muster, deren Beziehungen sich nach dem Übersetzen nicht mehr ändern. Die zweite Variante gehört zur Klasse der Muster, deren Beziehungen sich zur Laufzeit ändern können. Hinzu kommen Designvarianten, die in den informalen Teilen, der Implementierung oder dem Verhalten beschrieben werden.

Beispiel: Composite-Muster

Für die Erkennung von Entwurfsmusterinstanzen in Quelltexten sind die Beschreibung der Struktur, das Verhalten sowie insbesondere die Implementierung eines Musters von besonderer Bedeutung. Die informale Beschreibung eines Gamma-Musters und die dadurch entstehende Anzahl Varianten, die in existierenden Softwaresystemen vorhanden sind, soll anhand des Composite-Musters dargestellt werden.

Abbildung 3.2 zeigt die Struktur-Beschreibung des Composite-Musters als UML-Klassendiagramm. Das Composite-Muster ermöglicht eine Speicherung atomarer (Leaf) und zusammengesetzer Elemente (Composite). Die Kompositionsmöglichkeit wird durch die

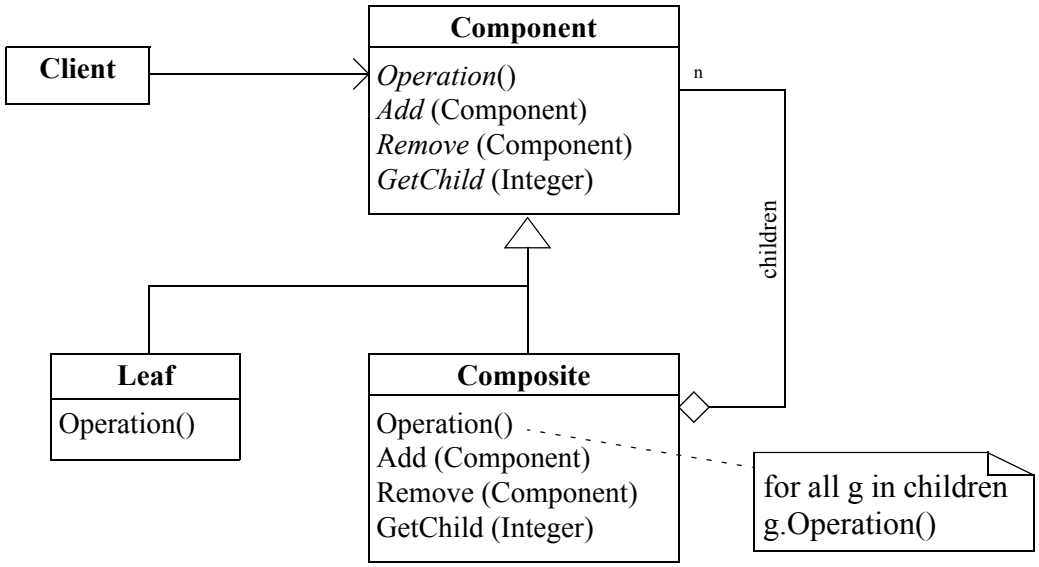


Abbildung 3.2 Struktur des Composite-Musters [GHJV95]

Assoziation children ausgedrückt, wobei Zugriffsmethoden verwendet werden, um Elemente einer Komposition hinzuzufügen (Add) oder zu löschen (Remove).

Der Zweck des Composite-Musters ist es, einen einheitlichen Zugriff auf die Elemente zu ermöglichen. Dies geschieht durch die Methode Operation in der Klasse Component, die für jedes Element, das in der Struktur vorhanden ist, überschrieben wird. Kompositionselemente leiten dabei den Aufruf an alle enthaltenen Elemente weiter. Dieses Verhalten wird durch den Kommentar an der Operation Methode der Klasse Composite festgelegt. Der Client dient als Verwalter und ist typischerweise die Verbindung zum System, in dem das Muster verwendet wird.

Im Gegensatz zu anderen Gamma-Mustern ist die Beschreibung des Musterverhaltens bereits teilweise in der Strukturbeschreibung durch den Kommentar enthalten und beschränkt sich im Kollaborationsteil auf: ”Clients use the Component class interface to interact with objects in the composite structure. If the recipient is a Leaf, then the request is handled directly. If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding”.

Abbildung 3.3 zeigt häufig vorzufindende Beispiele für Designvarianten des Composite-Musters. In der linken Variante sind die Klassen Component und Composite nicht von einander getrennt, wodurch einem Leaf ebenfalls Knoten hinzugefügt werden können, wenn die entsprechenden Methoden Add, Remove und GetChild nicht überschrieben werden.

Die Variante auf der rechten Seite, in der eine oder gegebenenfalls auch mehrere Klassen zwischen die Component- und die Composite-Klasse geschoben werden und verschiedene Blätter (LeafKind1, LeafKind2) unterschiedlicher Zwischenklassen (Intermediate) abgeleitet werden, wird häufig zur Unterscheidung der Blättergruppen eingeführt. Solche Strukturen finden sich häufig in Legacy-Systemen, in denen aufgrund der Evolution des Systems zur Verfeinerung der Blättergruppen neue Zwischenklassen eingebaut werden.

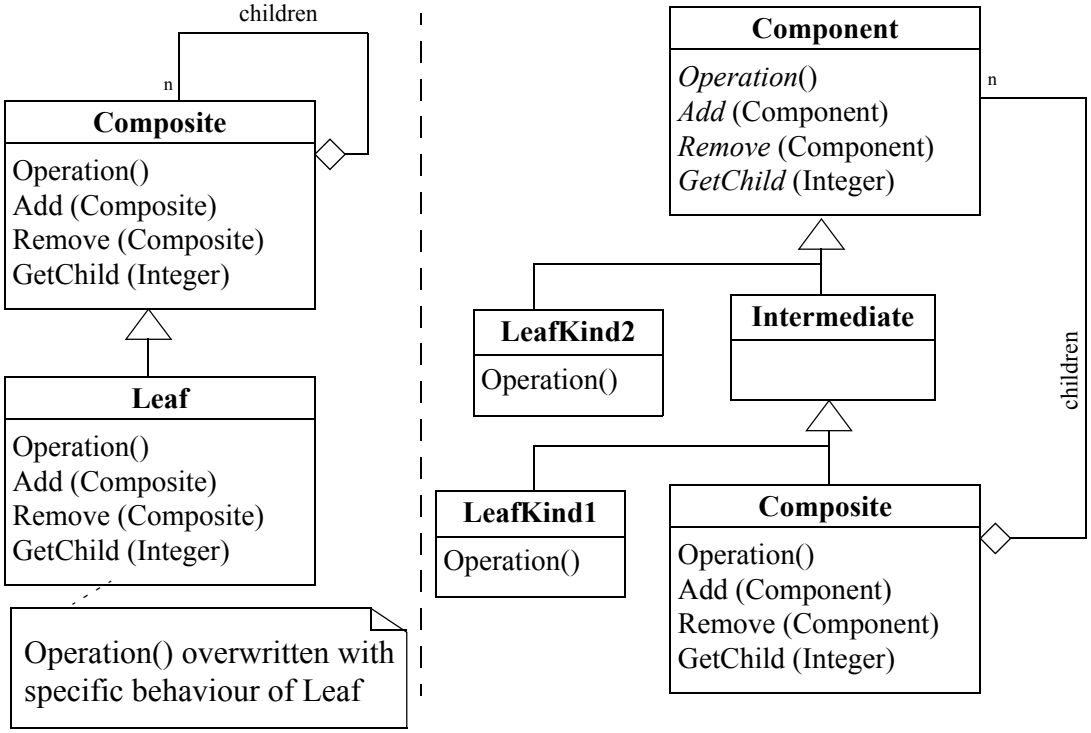


Abbildung 3.3 Variantenbeispiele des Composite-Musters

Zusätzlich zu den Designvarianten existieren zu einem Entwurfsmuster eine sehr große Zahl von Implementierungsvarianten. Implementierungsvarianten können sich zum Beispiel durch die Verwendung unterschiedlicher Schleifenkonstrukte unterscheiden. Ebenso können sich Implementierungsvarianten aufgrund der verwendeten Bibliotheken beziehungsweise zur Verfügung stehenden Schnittstellen unterscheiden. Auch die Verwendung anderer Muster wie zum Beispiel des Iterator-Musters im Gegensatz zum direkten Zugriff auf einen Container sind Unterscheidungsmerkmale verschiedener Implementierungsvarianten.

3.2 Analyseprozess

Der in dieser Arbeit vorgestellte Ansatz zur Erkennung von Entwurfsmusterinstanzen in Quelltexten großer Softwaresysteme besteht aus einem inkrementellen Analyseprozess, der in Abbildung 3.4 dargestellt ist.

Entwurfsmusterinstanzen werden im hier vorgestellten Ansatz durch eine Menge speziellen Graphersetzungsregeln, sogenannten Musterinstanzregeln, repräsentiert, die in einem Regelkatalog zusammengefasst werden. Die Anzahl der Musterinstanzregeln für Entwurfsmusterinstanzen eines Entwurfsmusters in einem Regelkatalog ist dabei beliebig. Jede Musterinstanzregel besitzt zusätzlich einen Wert, der die Genauigkeit der Musterinstanzregel bezogen auf die von ihr beschriebene Entwurfmusterinstanz angibt. Die Genauigkeitswerte der Musterinstanzregeln fließen in die Präzisionswerte der Ergebnisse der automatischen Analyse mit ein und unterstützen den Reengineer bei der Beurteilung der Ergebnisse.

Musterinstanzregeln werden auf der Grundlage einer abstrakten Syntaxgraphrepräsentation des zu analysierenden Quelltextes definiert. Der hier vorgestellte Ansatz ist dabei nicht an einen konkreten abstrakten Syntaxgraphen gebunden. Es können auch andere graphbasierte Repräsentationen eines Quelltextes wie Kontroll- oder Datenflüsse verwendet werden. Musterinstanzregeln und Regelkataloge werden in Kapitel 4 vorgestellt.

Aufgrund der meist informalen Entwurfsmusterbeschreibung existiert eine hohe Anzahl von Design- und Implementierungsvarianten von Entwurfsmustern in Softwaresystemen. Eine Implementierungsvariante entspricht dabei einer Entwurfsmusterinstanz. Alle Entwurfsmusterinstanzen durch Musterinstanzregeln eines Regelkatalogs zu beschreiben, ist aufgrund der hohen Anzahl nur theoretisch möglich. Allerdings ist die Anzahl unterschiedlicher Entwurfsmusterinstanzen in einem Softwaresystem stark eingeschränkt.

Einerseits werden einzelne Entwurfsmuster nur in bestimmten Systemen verwendet. Dies wird *Domänenabhängigkeit* genannt. Eine Textverarbeitung verwendet oftmals ein Composite-Muster zur internen Speicherung der Daten. Im Gegensatz dazu werden in ressourcenbeschränkten Applikationen wie zum Beispiel Java-Cards [HNS99] keine Entwurfsmuster verwendet, die unbegrenzt Ressourcen benötigen können. Es existieren außerdem Entwurfsmuster, die sich mit einzelnen Domänen beschäftigen. Zum Beispiel mit der Programmierung nebenläufiger Systeme [Lea97] oder für die Organisation reaktiver Systeme [CKV96].

Andererseits ist die Anzahl der Implementierungsvarianten eines Entwurfsmusters in einem Softwaresystem ebenfalls eingeschränkt. Ein Grund dafür ist, dass Entwickler unterschiedliche *Programmierstile* verwenden beziehungsweise sich an eine *Programmierrichtlinie* halten müssen. Das können zum Beispiel die Verwendung bestimmter Schleifenkonstrukte oder Namenskonventionen sein. Programmierrichtlinien, die für das gesamte zu analysierende Softwaresystem verwendet worden sind, liefern damit Informationen über Implementierungsvarianten. Aber auch die Kenntnis, welcher Entwickler welche Teile des Systems geschrieben hat, liefert Informationen über im System vorhandene Implementierungsvarianten.

Weiterhin werden Entwurfsmuster als Einheit beschrieben. Die *Komposition von Entwurfsmustern aus Teilmustern*, und damit die Möglichkeit zur Wiederverwendung schon gefundener Teilmuster, schränkt die Anzahl der Implementierungsvarianten im Software-

system zwar nicht ein, ist aber eine Optimierung bei ihrer Erkennung. Ein Beispiel für ein häufig wiederkehrendes Teilmuster ist das Delegation-Muster. Bei einem Delegation-Muster wird eine Aufgabe an eine andere Klasse beziehungsweise eine andere Methode delegiert. Im Beispiel des Composite-Musters - Abbildung 3.2 - enthält die Methode Operation der Klasse Composite ein Delegation-Muster oder kurz eine Delegation. In diesem Fall wird die Aufgabe der Methode eines Composite-Objekts an alle Kinder des Objekts delegiert.

Die Idee des hier vorgestellten Ansatzes ist, einen *initialen Regelkatalog* sukzessive an das zu analysierende Softwaresystem anzupassen, so dass er nur diejenigen Musterinstanzregeln enthält, die alle im System vorhandenen Entwurfsmusterinstanzen beschreiben [NSW+02]. Die Anpassung eines Regelkatalogs an ein zu analysierendes System erfolgt durch den Reengineer, der aufgrund der Ergebnisse einer inkrementellen automatischen Analyse die Musterinstanzregeln eines Regelkatalogs modifizieren kann. Die inkrementelle automatische Analyse versucht dabei, möglichst frühzeitig Entwurfsmusterinstanzen zu erkennen.

Für ein neu zu analysierendes Softwaresystem ist der erste Schritt des Reengineer im Analyseprozess, siehe Abbildung 3.4, die Auswahl eines geeigneten initialen Regelkatalogs. Kenntnisse über Programmierstile oder Stichproben des Quelltextes können den Reengineer bei der Auswahl eines Regelkatalogs unterstützen.

Der Reengineer hat im Folgenden die Möglichkeit, die Musterinstanzregeln mit Hilfe eines Editors zu modifizieren (regelnModifizieren). Es können neue Musterinstanzregeln hinzugefügt, Musterinstanzregeln gelöscht oder vorhandene Musterinstanzregeln verändert werden. Außerdem können die Genauigkeitswerte der Musterinstanzregeln geändert werden. Das Resultat der Modifikationen ist ein Regelkatalog, der vor der automatischen Analyse (analyseStarten) in einem Datencontainer abgelegt wird. Sollte noch kein Quelltext in Form eines abstrakten Syntaxgraphen im zweiten Datencontainer vorliegen, so wird der zu analysierende Quelltext durch einen Parser in den Datencontainer als abstrakter Syntaxgraph abgelegt. Sollte bereits aus einer vorherigen Prozessiteration ein annotierter abstrakter Syntaxgraph vorliegen, werden lediglich alle Annotationen gelöscht. Die beiden Datencontainer dienen dabei lediglich zur internen Verwaltung und stellen Schnittstellen zur Datenmanipulation zur Verfügung.

Die automatische Analyse des Quelltextes eines Softwaresystems erfolgt durch einen inkrementellen Regelausführungsmechanismus. Der Regelausführungsmechanismus wendet die Musterinstanzregeln des Regelkatalogs auf den abstrakten Syntaxgraphen des Quelltextes an. Musterinstanzregeln eines Regelkatalogs erzeugen bei einer erfolgreichen Anwendung einen neuen Knoten im abstrakten Syntaxgraphen. Diese Knoten annotieren andere Knoten durch entsprechende Kanten und werden deshalb Annotationsknoten genannt. Jeder Annotationsknoten stellt ein Ergebnis dar.

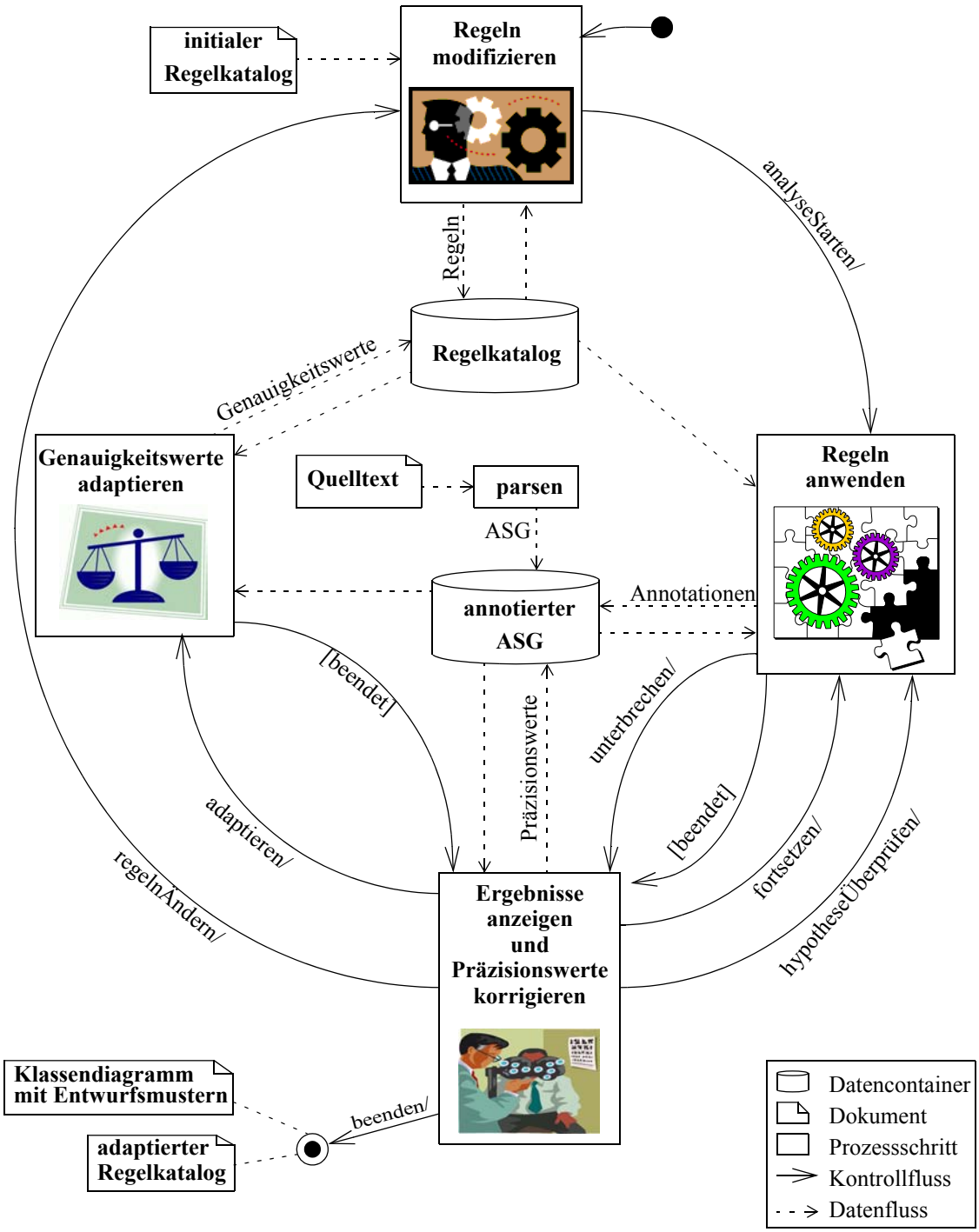


Abbildung 3.4 Analyseprozess

Relevante (Zwischen-)Ergebnisse der automatischen Analyse sind Annotationsknoten, die von Musterinstanzregeln erzeugt worden sind, die Entwurfsmusterinstanzen beschreiben. Der inkrementelle Regelausführungsmechanismus versucht dabei, diese relevanten Ergebnisse frühzeitig zu produzieren, im Gegensatz zu single-shot Ansätzen, die solche Ergebnisse erst zum Ende der automatischen Analyse liefern. Die Ausführung ist beendet, wenn keine Musterinstanzregel mehr angewendet werden kann. Die automatische Analyse wird in Kapitel 5 detailliert vorgestellt.

Große Softwaresysteme lassen sich im Allgemeinen in Teile mit geringerer Größe aufteilen. Dabei muss allerdings beachtet werden, dass dadurch Entwurfsmusterinstanzen über die verschiedenen Teile "verstreut" werden können. Prinzipiell müssen also für die Erkennung aller Entwurfsmusterinstanzen alle Teile des Softwaresystems analysiert werden. Insbesondere bei großen Systemen ist daher die frühzeitige Produktion relevanter Ergebnisse notwendig, damit der Reengineer gegebenenfalls in die Analyse eingreifen kann (unterbrechen).

Das (Zwischen-)Ergebnis des Regelausführungsmechanismus ist ein annotierter abstrakter Syntaxgraph und kann als annotiertes UML-Klassendiagramm angezeigt werden. Jeder Annotationsknoten - gefundene Entwurfsmusterinstanz - besitzt dabei einen Präzisionswert. Die Präzision errechnet sich aufgrund eines Genauigkeitswerts der Musterinstanzregel und der Präzision der Annotationsknoten, die bei der Regelanwendung beteiligt waren. Die Ergebnisse können vom Reengineer mit den entsprechenden Quelltextteilen verglichen und gegebenenfalls ihre manuell Präzision korrigiert werden. Die korrigierten Präzisionswerte werden direkt in den annotierten abstrakten Syntaxgraphen übernommen.

Nach der Beurteilung eines Zwischenergebnisses kann der Reengineer mit dem Regelausführungsmechanismus fortfahren (fortsetzen), Musterinstanzregeln modifizieren (regelnÄndern) oder die Genauigkeitswerte der Musterinstanzregeln automatisch anpassen lassen (adaptieren).

Wird die automatische Analyse fortgesetzt, wird der Regelausführungsmechanismus aufgrund seiner Inkrementalität auf dem letzten Zwischenergebnis mit den angepassten Präzisionswerten weiter ausgeführt. Dabei werden die Präzisionswerte der bereits erzeugten Annotationsknoten aufgrund der neuen Präzisionswerte der Musterinstanzregeln evaluiert. Anschließend werden neue Musterinstanzregeln angewendet.

Bei einer Änderung der Musterinstanzregeln startet der Regelausführungsmechanismus mit den geänderten Musterinstanzregeln (analyseStarten) wieder mit einem nicht annotierten abstrakten Syntaxgraphen des Quelltextes. Bisherige Zwischenergebnisse werden nicht weiter berücksichtigt.

Die Adaption der Genauigkeitswerte der Musterinstanzregeln erfolgt durch ein statistisches Verfahren auf Basis der korrigierten Präzisionswerte der Ergebnisse. Eine detaillierte Beschreibung der Adaption befindet sich in Kapitel 6. Die geänderten Genauig-

keitswerte der Musterinstanzregeln werden in den Regelkatalog übernommen. Nach der Adaption der Genauigkeitswerte kann der Reengineer weitere Korrekturen an den Präzisionswerten vornehmen.

Während der Anzeige eines (Zwischen-)Ergebnisses und der Korrektur der Präzisionswerte hat der Reengineer die Möglichkeit, den Analyseprozess zu beenden. Wieviele Iterationen im Prozess bei der Analyse eines Softwaresystems durchgeführt werden, ist also vom Reengineer abhängig, der entscheiden muss wann die produzierten Ergebnisse ausreichend sind. Das Ergebnis des Analyseprozesses ist einerseits ein Klassendiagramm annotiert mit Entwurfsmusterinstanzen und andererseits der auf das analysierte Softwaresystem angepasste Regelkatalog. Soll die Analyse zu einem späteren Zeitpunkt wieder aufgenommen werden, kann der Analyseprozess mit dem angepassten Regelkatalog als initialer Katalog wieder gestartet werden.

Der ausgewählte initiale Regelkatalog zu Anfang des Analyseprozesses kann die Anzahl der Prozessiterationen verringern, wenn die notwendigen Anpassungen gering sind. Andererseits kann ein falsch angepasster Regelkatalog zu einer Erhöhung der Prozessiterationen führen. So können zum Beispiel Kenntnisse über Programmierstile oder Stichproben des Quelltextes den Reengineer bei der Auswahl eines Regelkatalogs unterstützen. Alternativ zur Auswahl eines bereits angepassten Regelkatalogs kann auch ein kleiner Regelkatalog mit unpräzisen Musterinstanzregeln verwendet werden. Die Ergebnisse der automatischen Analyse können dann als Ausgangspunkt für eine bessere Wahl des initialen Katalogs dienen.

Während des Analyseprozesses hat der Reengineer die Möglichkeit, Hypothesen über existierende Entwurfsmusterinstanzen an einer bestimmten Stelle im Softwaresystem zu validieren (hypothesenÜberprüfen). Eine Hypothese wird durch die Erzeugung eines neuen Annotationsknotens vom Reengineer in den abstraken Syntaxgraphen eingebracht. Durch die sich anschließende automatische Analyse wird die Hypothese erhärtet oder revidiert werden. Der Ursprung einer Hypothese kann zum Beispiel eine (manuelle) Teilanalyse, ein Entwurfsdokument, Interviews mit ehemaligen Entwicklern oder die Erfahrung des Reengineer sein. Im Prinzip können Hypothesen auch automatisiert in ein Zwischenergebnis eingebaut werden, so dass zum Beispiel Entwurfsdokumente direkt Teil der Analyse werden können.

Außerdem können einerseits Zwischenergebnisse als Eingabe für andere Analysesysteme dienen, andererseits können Ergebnisse aus anderen Analysen in die Zwischenergebnisse einfließen. Ein Reengineer, der Hypothesen aufgrund seines Wissens in ein Zwischenergebnis einbringt und sie im weiteren Verlauf validieren lässt, ist dabei nur ein Beispiel. Weitere Analysen könnten automatisch aufgrund eines Zwischenergebnisses gestartet und die Ergebnisse im weiteren Verlauf verwendet oder validiert werden. Damit ist dies ein geeigneter Punkt für die Kopplung dieses Analyseprozesses mit anderen Analysen. Als Datenaustauschformat kann beispielsweise die Graph Exchange Language (GXL), die für den Austausch von Graphen allgemein entwickelt worden ist [Win01, WKR01], dienen.

Durch die Interaktionsmöglichkeiten des Reengineer lässt sich der Analyseprozess flexibel an das konkrete Vorgehen eines Reengineer anpassen. Das Vorgehen eines Reengineer hängt dabei von seinen Erfahrungen und Kenntnissen, aber auch von persönlichen Präferenzen ab. Zum Beispiel kann der Reengineer sich mit Hilfe eines Regelkatalogs, der unpräzise Musterinstanzregeln enthält, zu Anfang einer Analyse einen Überblick über die vorhandenen Entwurfsmusterinstanzen verschaffen, bevor er detailliertere Analysen vornimmt. Außerdem kann der Reengineer entscheiden, wann die Ergebnisse der Analyse ausreichend sind, oder sich bei der Analyse auf bestimmte Teile im Softwaresystem konzentrieren.

3.3 Ergebnis einer Analyse

Ergebnisse einer Analyse werden als annotiertes Klassendiagramm dargestellt. Dabei wäre auch eine andere Form der Präsentation eines Zwischenergebnisses denkbar. Klassendiagramme bieten sich allerdings in dem hier verwendeten Anwendungsbereich des Design-Recovery an.

Abbildung 3.5 zeigt das Ergebnis einer automatischen Analyse als annotiertes Klassendiagramm in UML ähnlicher Notation. Der zu analysierende Quelltext bestand aus drei

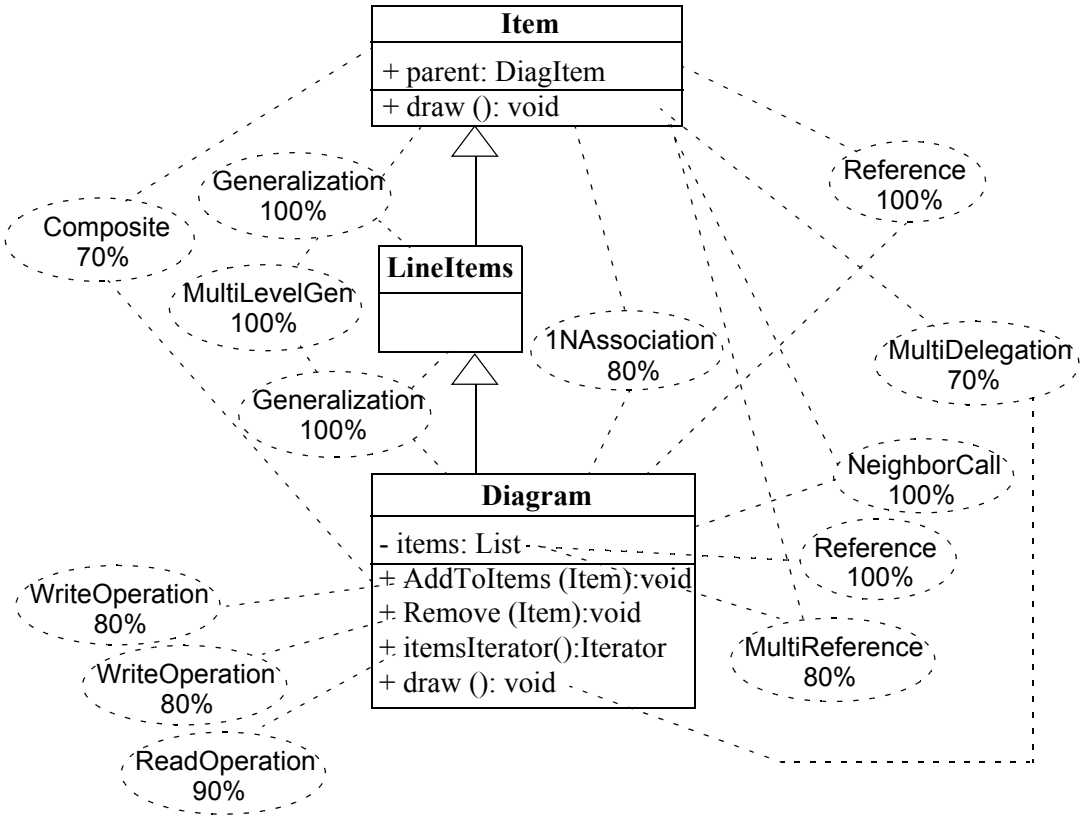


Abbildung 3.5 Beispiel eines Analyseergebnisses

Klassen, Item, LineItem und Diagram. Entsprechend besteht das Klassendiagramm aus drei Klassen, die durch verschiedene Annotationsknoten annotiert sind. Die Informationen, welche Klassen der Quelltext enthält und welche Vererbungsbeziehungen bestehen, sind direkt aus dem abstrakten Syntaxbaum abgeleitet worden. Ein solches Klassendiagramm wird als *rudimentäres* Klassendiagramm bezeichnet.

Im Gegensatz zu Vererbungsbeziehungen werden Verwendungsbeziehungen zwischen Klassen nicht aufgrund entsprechender Elemente im abstrakten Syntaxgraphen erzeugt. Dies liegt insbesondere daran, dass Verwendungsbeziehungen zwischen Klassen, wenn sie als Entwurfsmuster interpretiert werden, schon eine sehr hohe Anzahl an Implementierungsvarianten haben. Zum Beispiel werden bidirektionale Assoziationen von den gängigen Programmiersprachen wie C++ oder Java nicht unterstützt, sondern müssen durch Zeiger beziehungsweise Attribute und gegebenenfalls Zugriffsmethoden implementiert werden. Daher werden Verwendungsbeziehungen mit in den Regelkatalog aufgenommen, um sie flexibel anpassen zu können.

Welche Teile ein Knoten annotiert und welchen Präzisionswert der Knoten hat, ist direkt abzulesen. Der Annotationsknoten Composite (Mitte links) gehört dabei zu den relevanten Ergebnissen und besagt, dass der analysierte Quelltext eine Instanz des Gamma-Musters Composite beinhaltet. Beteiligte Klassen sind die Klasse Component und Composite. Das gefundene Gamma-Muster ist eine der Design-Varianten, die in Abbildung 3.3 vorgestellt worden sind und hat einen Präzisionswert von 70%. Die anderen Annotationsknoten sind Teil des Composite-Musters, wie zum Beispiel die 1NAssociation oder MultiDelegation Annotation. Andere Teilmuster annotieren Clichés.

Clichés sind stark programmiersprachenabhängige Implementierungsmuster. Eine Reference Annotation zum Beispiel besagt, dass sich in einer Klasse ein Attribut befindet, dessen Typ eine andere Klasse ist; also ein Zeiger. Die obere der beiden Reference Annotationen in Abbildung 3.5 markiert die direkte referentielle Beziehung der beiden Klassen *Item* und *Diagram*. Die untere Reference Annotation markiert die referentielle Beziehung zu einer Klasse, in der Objekte verwaltet werden; im Beispiel eine Liste.

ReadOperation und WriteOperation Annotationen kennzeichnen lesende beziehungsweise schreibende Methoden eines Attributs. Zum Beispiel müssen Zugriffsmethoden gemäß Java Beans Konvention mit get- beziehungsweise set- als Prefix des Attributnames deklariert sein. Aus der unteren Reference Annotation und den drei Zugriffsmethoden kann geschlossen werden, dass es sich um eine zu-N Beziehung zwischen der Klasse Diagram und Item handelt. Dies wird durch die Annotation MultiReference dargestellt. Aufgrund der MultiReference und der oberen Reference Annotation kann auf eine bidirektionale Assoziation zwischen den Klassen Diagram und Item geschlossen werden. Dies wird durch die 1NAssociation Annotation ausgedrückt.

Zusätzlich befinden sich in der Abbildung auch Annotationen, die eine Abstraktion beziehungsweise Verfeinerung einer anderen Annotation darstellen. Die NeighborCall Annotation entspricht einer Aufrufbeziehung, wohingegen eine MultiDelegation Annotation

Methodenaufrufe über alle in einem Container verwalteten Objekte markiert. Dass heißt, dass zu einer MultiDelegation Annotation auch immer eine NeighborCall Annotation mit den gleichen annotierten Objekten existiert. Umgekehrt gilt diese Regel allerdings nicht.

Eine weitere Besonderheit ist die Generalization Annotation, die eine direkte Korrespondenz zum Inheritance Objekt des abstrakten Syntaxgraphen, also der Vererbungsbeziehung zwischen zwei Klassen ist. Diese Annotationen werden durch die MultiLevelGeneralization Annotation so miteinander verknüpft, dass Vererbungsbeziehungen, die nicht direkt im abstrakten Syntaxgraphen abzulesen sind, ebenfalls annotiert werden. Im obigen Beispiel wird durch die MultiLevelGeneralization die Klasse Item als Superklasse und die Klasse Diagram als Subklasse annotiert.

3.4 Zusammenfassung

Der hier beschriebene Ansatz zur Erkennung von Entwurfsmusterinstanzen in Quelltexten großer Softwaresysteme nutzt die Tatsache aus, dass in einem konkreten Softwaresystem nur eine begrenzte Anzahl von Implementierungsvarianten eines Entwurfsmusters vorkommen. Dabei spielen bei der Analyse zum Beispiel Kenntnisse über verwendete Programmierrichtlinien oder persönliche Programmierstile der Entwickler, aber auch der Anwendungskontext des Softwaresystems eine Rolle.

Die Analyse eines Softwaresystems erfolgt durch einen inkrementellen Prozess, in den der Reengineer stark eingebunden ist. Frühzeitige Zwischenergebnisse der automatischen inkrementellen Analyse und Bewertung der Ergebnisse durch Präzisionswerte unterstützen den Reengineer dabei, die Analyse zu steuern.

Durch die Einbindung des Reengineer kann der Prozess flexibel für Analysen unterschiedlicher Systeme eingesetzt werden. Er lässt sich ebenfalls an persönliche Vorgehensweisen eines Reengineer anpassen.

Ebenso können Informationen, die durch andere Analysen erworben worden sind, manuell oder automatisch in die Analyse übernommen werden. Der Ansatz kann auf diese Weise mit anderen Analysesystemen gekoppelt werden.

KAPITEL 4: FORMALISIERUNG DER ENTWURFSMUSTERINSTANZEN

Der im vorherigen Kapitel vorgestellte Ansatz zur Erkennung von Entwurfsmusterinstanzen in großen Softwaresystemen besteht aus einem semi-automatischen Analyseprozess, der eine inkrementelle automatische Analyse des Quelltextes integriert. Für die automatische Analyse ist eine Formalisierung der Entwurfsmusterinstanzen notwendig.

In diesem Kapitel wird die Formalisierung der Entwurfsmusterinstanzen vorgestellt. Entwurfsmusterinstanzen werden als Graphtransformationsregeln [Roz99] - so genannte *Musterinstanzregeln* - auf einer abstrakten Syntaxgraphrepräsentation des zu analysierenden Systems definiert. Eine Menge von Musterinstanzregeln werden in einem Regelkatalog zur Erkennung der Entwurfsmusterinstanzen zusammengefasst.

Im Folgenden wird zuerst ein Beispielkatalog vorgestellt. In den sich anschließenden Abschnitten werden abstrakte Syntaxgraphen und Regelkataloge aufbauend auf der Definition eines objektorientierten Graphen aus [Zün01] formalisiert. Die Semantik eines Regelkatalogs wird durch die Abbildung der einzelnen Musterinstanzregeln auf *Story-Pattern* [Zün01], eine spezielle Art von Graphtransformationsregeln, definiert. Abschließend werden Erweiterungen der Musterinstanzregeln vorgestellt.

4.1 Beispiel

Musterinstanzregeln werden auf Basis eines objektorientierten abstrakten Syntaxgraphen des Quelltextes definiert. Abstrakte Syntaxgraphen repräsentieren sowohl Klassen, Attribute und Methodendeklarationen sowie Methodenrümpfe des Quelltextes. Insbesondere Methodenrümpfe sind bei der Erkennung beziehungsweise Unterscheidung der Entwurfsmusterinstanzen notwendig, da sich Entwurfsmuster teilweise nur aufgrund ihres Verhaltens unterscheiden.

Prinzipiell können die Struktur und das Verhalten eines Entwurfsmusters durch UML-Diagramme wie zum Beispiel Klassendiagramme, Aktivitätsdiagramme, Kollaborationsdiagramme oder Statecharts beschrieben werden. Eine solche Beschreibung würde dazu führen, dass alle möglichen Implementierungsvarianten aus den Beschreibungen für eine Erkennung im Quelltext erzeugt werden müssten. Aufgrund der hohen Anzahl Implementierungsvarianten ergäbe sich eine hohe Anzahl Beschreibungen für die Erkennung im Quelltext. Die hohe Anzahl Beschreibungen würde eine Erkennung insbesondere bei großen Softwaresystem fehlschlagen lassen.

Das folgende Beispiel ist ein einfacher Regelkatalog mit drei Musterinstanzregeln für strukturelle Teile eines Composite-Musters, siehe Abbildung 3.3. Die dynamischen An-

teile des Entwurfsmusters werden in diesem einleitenden Beispiel nicht mit berücksichtigt. Der Regelkatalog ist in Abbildung 4.1 dargestellt und umfasst drei Musterinstanzregeln mit Namen Composite (rechts oben), Generalization (rechts unten) und MultiLevelGen (links unten) sowie ein Regeldefinitionsdiagramm (links oben). Das Regeldefinitionsdiagramm ist als UML Klassendiagramm, und die Musterinstanzregeln sind als UML Kollaborationsdiagramme dargestellt.

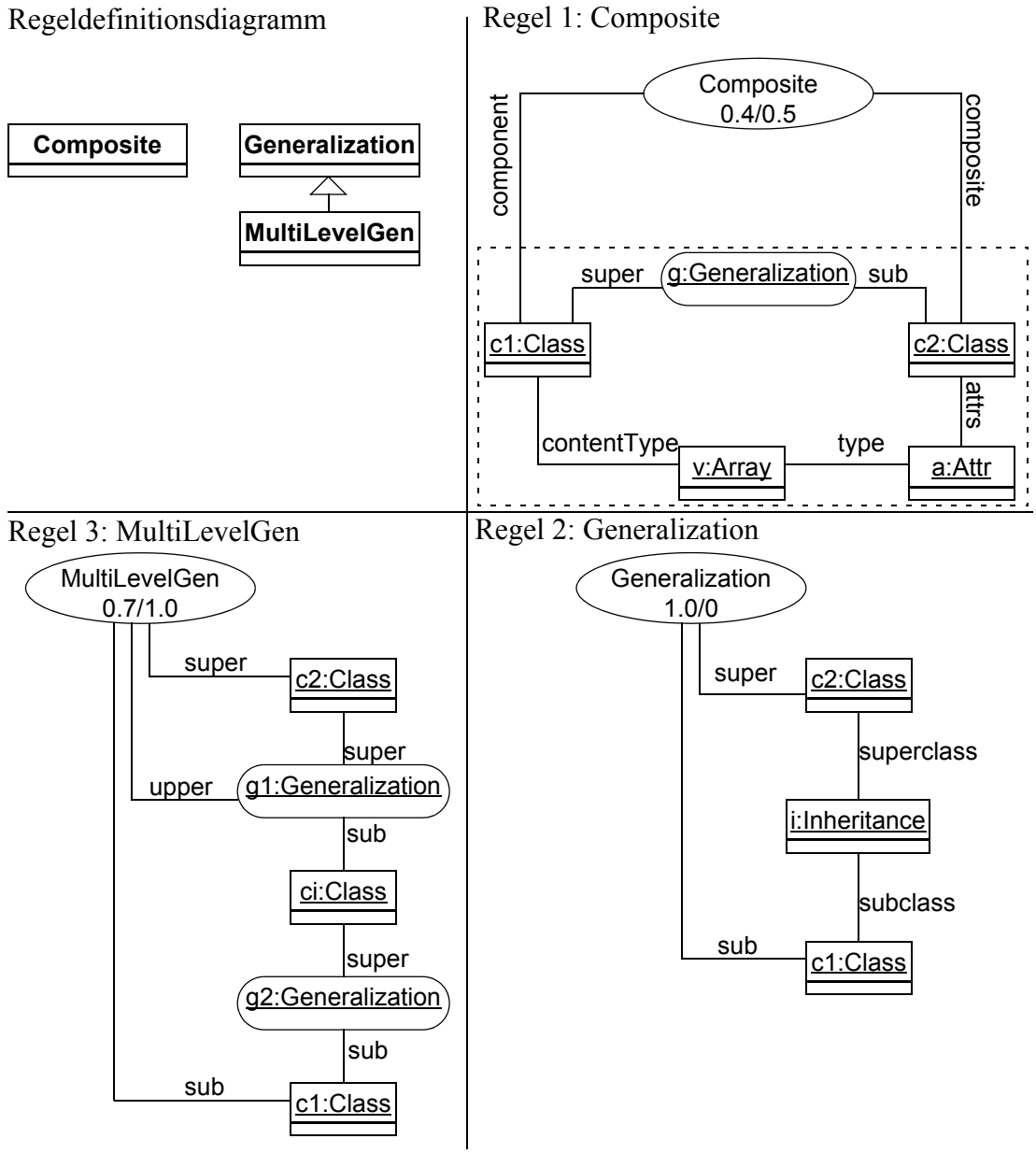


Abbildung 4.1 Regelkatalogbeispiel

Musterinstanzregeln bestehen aus einem Ausschnitt des abstrakten Syntaxgraphen. In der Composite-Regel ist der Ausschnitt durch eine gestrichelte Linie markiert. Im Folgenden werden diese Ausschnitte einer Musterinstanzregel als *RegelASGausschnitte* bezeichnet.

RegelASGausschnitte enthalten zwei Arten von Knoten. Knoten des abstrakten Syntaxgraphen werden als UML Objekte mit Namen und Typ dargestellt. Die anderen Knoten sind so genannte *Annotationsknoten*. Sie sind UML Objekte mit dem Stereotyp «Annotation» und werden mit abgerundeten Ecken dargestellt. Beziehungen zwischen den Knoten werden durch ihre Namen identifiziert.

Bezogen auf Graphtransformationsregeln entspricht der RegelASGausschnitt der linken Regelseite, und alle Knoten und Beziehungen entsprechen der rechten Regelseite. Die Semantik einer Graphtransformationsregel ist, die linke Regelseite im annotierten abstrakten Syntaxgraphen, zu finden, und durch die rechte Regelseite zu ersetzen. Im Beispiel der Composite-Regel wird ein abstrakter Syntaxgraphausschnitt gesucht, der zwei Klassen c1:Class, c2:Class enthält, die über eine Arrayreferenz a:Attr, v:Array miteinander in Beziehung stehen. Der Annotationsknoten g:Generalization drückt aus, dass eine Vererbungsbeziehung zwischen den beiden Klassen existiert. Sollte ein solcher Ausschnitt im Quelltext gefunden werden, wird ein neuer Annotationsknoten :Composite erzeugt und mit den beiden Klassen verbunden. Der Annotationsknoten g:Generalization wurde dabei schon zuvor von der entsprechenden Generalization-Regel erzeugt. Damit ist es möglich, dass Musterinstanzregeln aus anderen Musterinstanzregeln komponiert werden können.

Weiterhin gehört zu jeder Musterinstanzregel ein *Musterinstanzknoten* der als Ellipse dargestellt wird. Dieser Knoten bezeichnet den Namen der Musterinstanzregel und beinhaltet zwei Werte, den *Genauigkeitswert* und den *Schwellwert*. Der Genauigkeitswert gibt die Genauigkeit der Musterinstanzregel bezogen auf die von ihr beschriebene Entwurfsmusterinstanz an. Er sollte so gewählt werden, dass er das Verhältnis der richtigen Ergebnisse zu allen gefundenen Ergebnissen bei der Anwendung der Musterinstanzregel auf einen abstrakten Syntaxgraphen beschreibt. Bei der Ausführung einer Musterinstanzregel dient der Schwellwert als unterer Grenzwert, wodurch Ergebnisse mit einem Präzisionswert unterhalb des Schwellwerts bei der weiteren Analyse nicht weiter betrachtet werden. Zusätzlich besitzt der Musterinstanzknoten noch Beziehungen zu Knoten im Ausschnitt des abstrakten Syntaxgraphen.

Zusätzlich zur Komposition kann das *Regeldefinitionsdiagramm* verwendet werden, um alternative Annotationen bei der Suche eines Abstraktsyntaxgraphausschnitts zu verwenden. Das Regeldefinitionsdiagramm enthält jeweils eine Klasse für jede im Katalog verwendete Musterinstanzregel und zusätzlich Verfeinerungsbeziehungen wie zum Beispiel zwischen Generalization und MultiLevelGen. Die Verfeinerung entspricht einer strukturellen Vererbung im objektorientierten Sinn. Durch Polymorphie kann so zum Beispiel anstatt der Generalization-Annotation in der Composite-Regel alternativ eine MultiLevelGen-Annotation bei der Ausführung der Musterinstanzregel verwendet werden. Die MultiLevelGen-Regel annotiert Vererbungen zwischen zwei Klassen, die allerdings nicht

direkt über eine Vererbungsbeziehung verbunden sind. Direkte Vererbungen zwischen zwei Klassen werden durch die Generalization-Regel annotiert. Bezogen auf die Varianten des Composite-Musters in Abbildung 3.3 werden durch die MultiLevelGen-Regel alle dargestellten Musterinstanzen des Composite-Musters erkannt.

Die Kombination von Komposition und Vererbung bildet einen flexiblen Mechanismus zur Definition von Regeln für Entwurfsmusterinstanzen. Es kann, wie im obigen Beispiel der Generalization-Regel und MultiLevelGen-Regel, Rekursion ausgedrückt werden oder einfache Alternativen mit Hilfe von “abstrakten” Musterinstanzregeln und “konkreten” Musterinstanzregeln definiert werden.

Ein Regelkatalog muss folgende Konsistenzbedingungen einhalten:

- Die Regelnamen in einem Regelkatalog sind eindeutig. Es gibt also keine zwei Klassen im Regeldefinitionsdiagramm mit gleichem Namen.
- Jedem Regelnamen ist eine Musterinstanzregel zugeordnet, deren Musterinstanzknoten den Regelnamen enthält. Der Musterinstanzknoten (Objekt) kann als Instanz der entsprechenden Klasse angesehen werden.
- Jeder Knoten in einer Musterinstanzregel ist Instanz entweder einer Klasse des abstrakten Syntaxgraphen gemäß Schemadefinition oder Instanz einer Klasse im Regeldefinitionsdiagramm.
- Musterinstanzregeln besitzen einen zusammenhängenden RegelASGausschnitt. Bezogen auf Graphtransformationsregeln heißt das, dass die linke Regelseite ein zusammenhängender Graph sein muss. Diese Eigenschaft wird bei der Regelausführung zur Optimierung Regelausführungsreihenfolge verwendet. In der Praxis hat sich herausgestellt, dass dies keine Einschränkung ist, weil bei der Erkennung von Entwurfsmusterinstanzen immer ein Knoten in die Musterinstanzregel mit aufgenommen werden kann, so dass der RegelASGausschnitt zusammenhängend ist.
- Musterinstanzregeln beziehungsweise der Musterinstanzknoten einer Musterinstanzregel und die Beziehungen zu Objekten des RegelASGausschnitts werden als Deklaration angesehen. Werden Annotationsknoten, die von einer Regel erzeugt werden in einer Deklaration einer anderen Musterinstanzregel verwendet, dürfen nur Beziehungen verwendet werden, die in der Deklaration zur Annotation gehörenden Musterinstanzregel vorkommen. Dabei müssen Namen eindeutig sein, und das Ziel der Beziehung muss, ausgehend von der Annotation, mit der Deklaration ebenfalls übereinstimmen. Im Beispiel der Composite-Regel dürfen die Namen der Beziehungen, die vom g:Generalization Knoten ausgehen, nur sub und super, sein und die Ziele der Beziehungen müssen jeweils Objekte vom Typ Class sein.
- Bei der Verfeinerungsbeziehung muss die Kindregel mindestens alle Beziehungen bereitstellen, die in der Vaterregel deklariert sind. Beziehungen können also nicht redefiniert werden. Es können lediglich zusätzliche Beziehungen verwendet werden.

Erweiterte Sprachkonstrukte zur Verwendung in Musterinstanzregeln werden in Abschnitt 4.5 vorgestellt. Dafür werden zuerst abstrakte Syntaxgraphen auf Basis eines objektorientierten Graphmodells formalisiert. Anschließend werden die Regelkataloge aufbauend auf den abstrakten Syntaxgraphen definiert und ihre Semantik durch die Abbildung auf Story-Pattern [Zün01] vorgestellt.

4.2 Graphrepräsentation des Quelltextes

In diesem Ansatz wird ein annotierbarer abstrakter Syntaxgraph des Quelltextes verwendet, der auf einem wohlgeformten, objektorientierten Graphmodell definiert ist. Das Graphmodell ist mit dem der objektorientierten Programmiersprachen wie zum Beispiel Java vergleichbar und ist Basis für das in [Zün01] vorgestellte formale Regelwerk. Das Graphmodell bildet die Grundlage für die Definition von Musterinstanzregeln und wird daher im Folgenden beschrieben.

Basierend auf dem Graphmodell besteht ein annotierbarer abstrakter Syntaxgraph (ASG) aus einer Schemainformation (einem Schema) und einer Ausprägung, die konform zum Schema (wohlgeformt) sein muss[1]. Die Eigenschaft der Objektorientierung bezieht sich auf Vererbungsbeziehungen in der Schemainformation und der darauf aufbauenden Polymorphieeigenschaft. Die Eigenschaft der Annotierbarkeit besagt, dass es möglich ist, das Schema eines ASG derart zu erweitern, dass zusätzliche Knoten (Annotationen) in der Ausprägung erzeugt werden können, die dann konform zur geänderten Schemainformation sind.

Die Beschreibung eines annotierbaren abstrakten Syntaxgraphen und die formale Beschreibung des verwendeten wohlgeformten, objektorientierten Graphmodells ist ein Teil dieses Abschnitts. Des Weiteren wird gezeigt, wie aus einer gegebenen Grammatik ein entsprechendes Schema erzeugt werden kann, das die Grundlage für die Definition von Musterinstanzregeln zur Erkennung von Entwurfsmusterinstanzen ist.

4.2.1 Grammatiken und Abstrakte Syntaxbäume

Abstrakte Syntaxgraphen werden vorwiegend im Übersetzerbau verwendet [ASU86]. Typischerweise enthalten abstrakte Syntaxgraphen einen abstrakten Syntaxbaum, der aus einer Grammatik erzeugt werden kann.

```
1: class House {
2:    i : int;
3:    void m () {
4:       i = 2;
5:    }
6: }
```

Abbildung 4.2 Quelltextbeispiel

1. Im Allgemeinen wird die Ausprägung eines Graphen als abstrakter Syntaxgraph bezeichnet.

Die Grammatikregeln in Abbildung 4.3 sind vereinfachte Regeln einer Java Grammatik. Nichtterminale sind *{CLASS, BODY, ATTR, METHOD, METHODBODY, ASSIGNMENT}*. Die Menge der Terminale besteht aus den Schlüsselwörtern *{"class", "int", "void"}*, den Separatoren *{"{", "}", "()", "=", ";", ":"},* der Menge der Bezeichner *Identifier*, der Menge der Literale *Literal* und der Menge der Datentypen *Type = {"int", " void", "bool"}*. Die Menge der Bezeichner und Literale wird durch eine lexikalische Analyse ermittelt. Für dieses Beispiel ist *Identifier = {"i", "m"}* und *Literal = {"2"}*.

CLASS → class Identifier { BODY }*
BODY → (ATTR, METHOD)
ATTR → Identifier : Type ;
METHOD → Type Identifier () { METHODBODY }
*METHODBODY →(ASSIGNMENT)**
ASSIGNMENT → Identifier = Literal ;

Abbildung 4.3 Grammatikregelbeispiel

Die Schemainformation für den abstrakten Syntaxgraphen ist in Abbildung 4.4 als UML-Klassendiagramm dargestellt. Für jedes Nichtterminal ist eine eigene Klasse, zum Beispiel Class, Attr oder Method angelegt worden. Dabei sind Vererbungsbeziehungen für Regeln erzeugt worden, die nur alternative Nichtterminale enthalten, die wiederum nur in einer Regel verwendet werden (2. Regel).

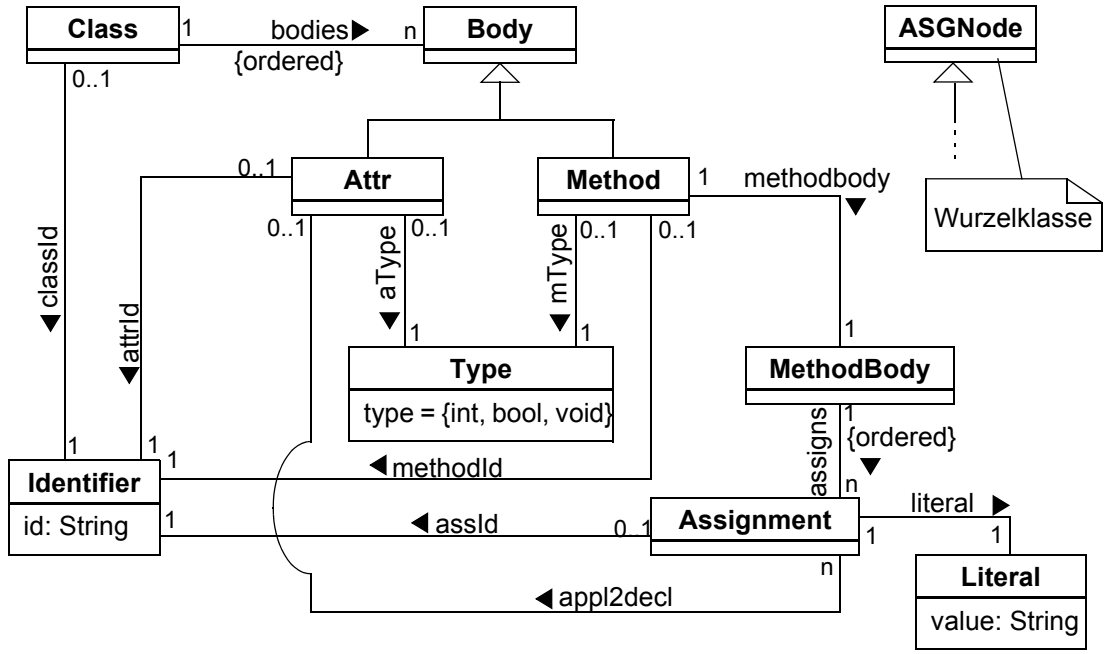


Abbildung 4.4 Schemabeispiel für erzeugten abstrakten Syntaxgraph

Die Assoziationen mit ihren entsprechenden Kardinalitäten zwischen den Klassen werden ebenfalls aufgrund der Grammatik erzeugt, zum Beispiel die Assoziation classId zwischen der Klasse Class und Identifier. Wenn die Kardinalität nicht beschränkt ist, wird eine geordnete Assoziation verwendet. Die Klassen Identifier, Type und Literal besitzen jeweils ein Attribut, das den jeweiligen Wert speichert.

Für Terminale, die konkrete Syntax repräsentieren, werden keine Klassen angelegt. Entweder sind sie Separatoren, die in der graphischen Repräsentation durch entsprechende Beziehungen ausgedrückt werden, oder sie werden durch entsprechende Klassen von Nichtterminalen repräsentiert. Zum Beispiel ist das Terminal *"class"*, das durch das Nichtterminal CLASS (1. Regel) und damit durch die Klasse Class repräsentiert wird, ein Separator. Damit entspricht der Teil der Schemainformation, der aus der Grammatik hergeleitet wird, der Schemainformation für einen abstrakten Syntaxbaum.

In Abbildung 4.4 wird der Anteil der Schemainformation für einen abstrakten Syntaxbaum durch die zusätzliche appl2decl-Assoziation erweitert. Diese Assoziation verbindet die Klasse Assignment und die Klasse Attr, wodurch ein abstrakter Syntaxgraph entsteht. Im abstrakten Syntaxgraphen bedeutet dies, dass die Variable, die einen neuen Wert zugewiesen bekommt, als Attribut deklariert ist. Solche semantischen Informationen können nicht durch kontextfreie Grammatikregeln wie oben ausgedrückt werden, sondern müssen durch weitere Analysen beziehungsweise kontextsensitive Grammatikregeln definiert werden.

4.2.2 Schemainformation eines Graphen

Für die Formalisierung von annotierbaren abstrakten Syntaxgraphen die Definition eines wohlgeformten objektorientierten Graphen aus [Zün01] verwendet. Die Darstellung einer Schemainformation erfolgt durch UML Klassendiagramme, siehe Abbildung 4.4.

Die Schemainformation besteht aus einer Menge von Knotennamen *NL*, Kantennamen *EL* und Attributbezeichnern *A*. Die Knotennamen entsprechen Klassen in UML-Klassendiagrammen. Aufgrund der Mengendarstellung müssen alle Bezeichner eindeutig sein. Es dürfen also zum Beispiel keine zwei Attribute in unterschiedlichen Klassen den gleichen Bezeichner haben. Dies ist eine Einschränkung, die aber durch entsprechende Umbenennungen einfach umgesetzt werden kann. Im weiteren Verlauf sind daher doppelte Bezeichner in UML Klassendiagrammen erlaubt, so lange sie eindeutig umbenannt werden können. Zum Beispiel wird Attributbezeichnern immer der Klassenname vorangestellt, wodurch zwei Klassen ein Attribut mit gleichem Namen haben können.

Attribute werden durch die Funktion *Attrs* einer Klasse zugewiesen und werden gleichzeitig typisiert. Attribute können dabei nur Basisdatentypen der UML wie *Boolean, Integer, String, Float, ...* und so weiter sein. Die Wertebereiche der Basisdatentypen entsprechen ebenfalls der UML.

Definition 4.1: Schemainformation eines objektorientierten Graphen

Die Schemainformation eines objektorientierten Graphen ist gegeben durch
SI := (NL, EL, A, IsAs, Assocs, Attrs) mit:

- *NL* ist eine endliche Menge von Knoten(bezeichnern) Klassen
- *EL* ist eine endliche Menge von Kanten(bezeichnern)
- *A* ist eine endliche Menge von Attributbezeichnern
- *Attrs:* $A \rightarrow NL \times BaseTypes$ *ist total und* Attribute
  BaseTypes = Basisdatentypen der UML
- *IsAs* $\subseteq NL \times NL$ *und ist zyklenfrei* Vererbungen
- *Assocs:* $EL \rightarrow NL \times MulInf \times \wp(aTypes) \times BaseTypes \times NL \times MulInf$ *mit*
  MulInf := {one, many} Assoziationen
  aTypes := {qualified, aggregation, ordered}

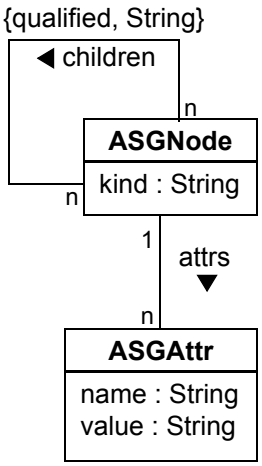
Die Relation *IsAs* beschreibt Vererbungsbeziehungen zwischen zwei Klassen. Die Vererbungsbeziehungen müssen dabei zyklenfrei sein. Dabei wird $(n1, n2) \in IsAs$ beschrieben als *n1 isA n2*. Die Definition der Schemainformation eines objektorientierten Graphen lässt Mehrfachvererbung zu, allerdings wird im weiteren Verlauf dieser Arbeit nur eine einfache Vererbung verwendet.

Verwendungsbeziehungen, so genannte *Assoziationen*, werden durch die partielle Funktion *Assocs* beschrieben. Eine Assoziation ist beschrieben als *Assocs (el) = (srcNI, srcCard, atype, abasetype, tgtNI, tgtCard)*. Der Name einer Assoziation ist der zugehörige Kantenbezeichner. Assoziationen sind Beziehungen zwischen zwei Knoten *NL*, wobei die Enden der Assoziation als Rollen bezeichnet werden. Die Rollen besitzen Kardinalitäten *one* oder *many*, wobei die Kardinalität nur eine obere Schranke darstellt.

Assoziationen können vom Typ *qualified*, *aggregation* oder *ordered* sein. Der *qualified* Typ sagt aus, dass die assoziierten Objekte zur Laufzeit durch einen Schlüssel erreichbar sind. Der Typ des Schlüssels ist als UML Basisdatentyp gegeben[1]. Bei Assoziationen vom Typ *aggregation* werden alle assoziierten Objekte beim Löschen des aggregierenden Objekts ebenfalls gelöscht. Der Typ *ordered* einer Assoziation sagt aus, dass assoziierte Objekte geordnet sind. Die Leserichtung einer Assoziation ist von links nach rechts. Dies sagt dabei nichts über die Navigierbarkeit der Beziehung aus. Assoziationen sind immer *bi-direktional*.

Als Beispiel für eine Schemainformation in Abbildung 4.5 eine allgemeine Schemainformation eines abstrakten Syntaxgraphen dargestellt. Abstrakte Syntaxgraphen sind gerich-

1. Bei Assoziationen, die nicht vom Typ qualified sind, wird der Typ des Schlüssels ignoriert. Dies erspart die Behandlung von einigen Sonderfällen und lässt sich technisch nutzen, um bei Änderungen des Assoziationstyps den Typ des Schlüssels gegebenenfalls später zu rekonstruieren.



- NL := {ASGNode, ASGAttr}
- EL := {children, attrs}
- A := {kind, name, value}
- Attrs (kind) := (ASGNode, String)
- Attrs (name) := (ASGAttr, String)
- Attrs (value) := (ASGAttr, String)
- IsAs := ∅
- Assocs (children) := (ASGNode, many, {qualified}, String, ASGNode, many)
- Assocs (attrs) := (ASGNode, one, ∅, String, ASGAttr, many)

Abbildung 4.5 Schemabeispiel für allgemeine abstrakte Syntaxgraphen

tete, attributierte, knoten- und kantenmarkierte Graphen. Die Schemainformation ist sowohl als Klassendiagramm als auch gemäß Definition 4.1 dargestellt.

Die Knotenmarkierung wird durch das kind Attribut der Klasse ASGNode erreicht, und die Kantenmarkierung wird durch die qualifizierte children-Assoziation ermöglicht. Die Attributierung einzelner Knoten wird durch die Klasse ASGAttr erreicht, die einen Namen name und einen Wert value beinhaltet.

In der Praxis wird für einen abstrakten Syntaxgraphen im Allgemeinen nicht die vorgestellte Schemainformation, sondern es werden verfeinerte Schemainformationen wie in Abbildung 4.4 verwendet. Diese verfeinerten Schemainformationen lassen sich allerdings auf die Schemainformation aus Abbildung 4.5 abbilden.

4.2.3 Ausprägung eines Graphen

Der abstrakte Syntaxgraph für das Quelltextbeispiel aus Abbildung 4.2 ist in Abbildung 4.6 als UML-Kollaborationsdiagramm dargestellt, siehe [Zün01].

Der Anteil des abstrakten Syntaxbaums für den Quelltext ist durch die Knoten (Objekte) und die vertikal verlaufenden Kanten (Links) repräsentiert. Der appl2Decl-Link ist in diesem Beispiel der einzige nicht zum abstrakten Syntaxbaum gehörende Teil.

Objekte besitzen in diesem Beispiel keinen Bezeichner, sondern es sind lediglich die Klassennamen dargestellt. Eindeutige Bezeichnernamen können entweder automatisch erzeugt oder explizit vergeben werden. Beziehungsnamen entsprechen Assoziationen in der Schemainformation, wobei teilweise die Beziehungsnamen erst durch Hinzufügen einer Nummer eindeutig werden. Die Objekte :Identifier, :Type, :Literal besitzen als Attribut Werte der lexikalischen Analyse.

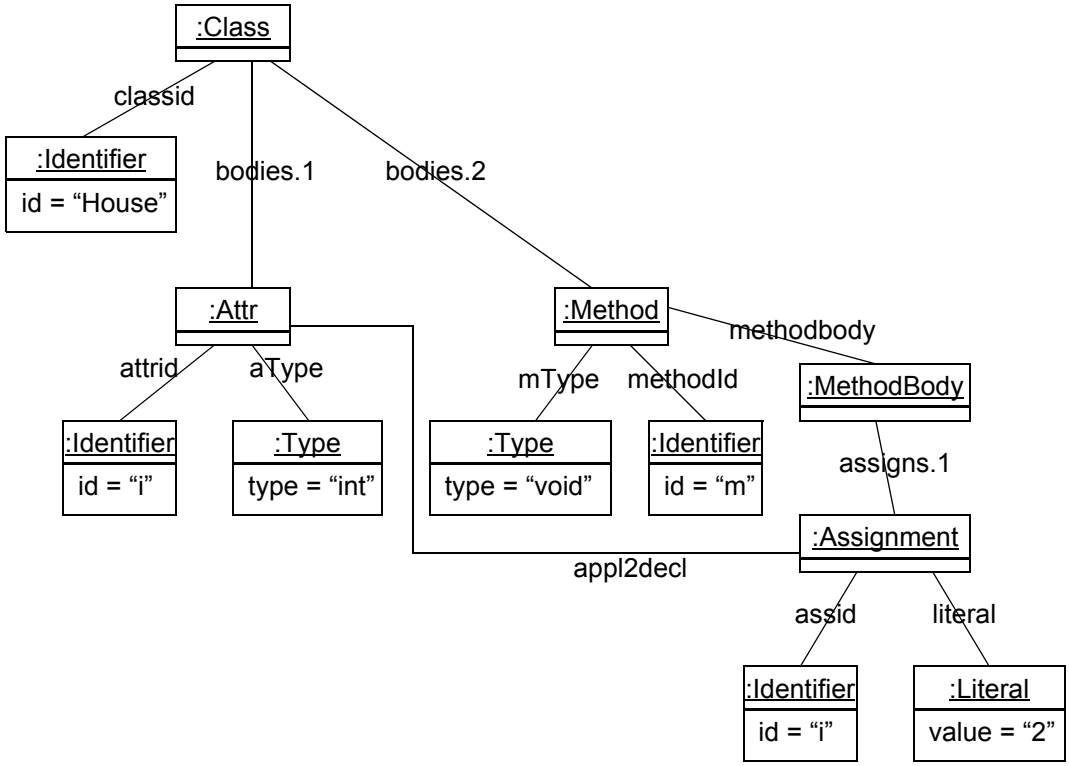


Abbildung 4.6 Abstrakter Syntaxgraph für Beispielquelltext

UML Kollaborationsdiagramme umfassen eine Reihe weiterer Sprachelemente wie zum Beispiel Methodenaufrufe und Kontrollfluss. Zur Darstellung einer Ausprägung werden allerdings nur Objekte und Links verwendet. Alternativ können auch UML Objektdiagramme verwendet werden.

Definition 4.2: Ausprägung eines objektorientierten Graphen

Eine Ausprägung einer Schemainformation *SI = (NL, EL, A, IsAs, Assocs, Attrs)* ist gegeben durch *Ext := (N, E, inst, av)* mit:

- N ist eine endliche Menge von Knoten(identifikatoren) — Objekte
- $E \subseteq N \times EL \times \{\{\varepsilon\} \cup \mathbb{R}\} \times \{\{\varepsilon\} \cup AttrVal\} \times N$ *mit* — Links
- *inst:* $N \rightarrow NL$ *ist total* — Instanzfunktion
- *av:* $N \times A \rightarrow AttrVal$ — Attributwerte

Die Ausprägung einer Schemainformation ist definiert durch eine Menge von Knoten N (Objekte). Knoten besitzen dadurch eine eindeutige Identifikation. Kanten (Links) werden durch die Relation E beschrieben. Eine Kante besteht zwischen zwei Knoten N und gehört zu einer Assoziation aus der Schemainformation *EL*. Ordnungsnummern werden durch die Menge $\{\{\varepsilon\} \cup \mathbb{R}\}$ bezeichnet, und $\{\{\varepsilon\} \cup AttrVal\}$ ist ein Qualifizierer des

Links. Der Wertebereich ist wie bei Attributen durch die UML Basistypen festgelegt. Ein Element der Relation wird geschrieben als *e = (src, el, i, q, tgt)*.

Die Funktion *inst* beschreibt den Zusammenhang zwischen Objekten und Klassen und wird als "Instanz von" gelesen. Die Funktion ist total, so dass jedes Objekt Instanz einer Klasse der Schemainformation ist.

Die Funktion *av* weist den Attributnamen der Schemainformation für ein Objekt einen Wert *AttrVal* zu. Der Wertebereich ist durch die UML Basisdatentypen der Attribute definiert.

4.2.4 Objektorientierter Graph

Die Schemainformation aus Definition 4.1 und die Ausprägung einer Schemainformation aus Definition 4.2 bilden einen objektorientierten Graphen.

Definition 4.3: Objektorientierter Graph

Ein Graph *G := (SI, Ext)* mit einer Schemainformation *SI* und einer Ausprägung *Ext* ist ein objektorientierter Graph.

In einem objektorientierten Graphen kann ein Objekt Instanz einer Klasse sein, wenn es einen direkten Zusammenhang durch die Funktion *inst* der Ausprägung oder durch einen transitiven Zusammenhang aufgrund der Vererbungsbeziehungen der Klassen gibt. Definition 4.4 stellt diese Instanzbeziehung dar.

Definition 4.4: Objekt ist Instanz einer Klasse

Gegeben ist ein objektorientierter Graph *G := (SI, Ext)* mit Schemainformation *SI := (NL, EL, A, IsAs, Assocs, Attrs)* und Ausprägung *Ext := (N, E, inst, av)*.
Dann ist die Funktion *instanceOf : (N, NL)* $\rightarrow$ *boolean* definiert als:

- *instanceOf (o* $\in$ *N, c* $\in$ *NL) =*
  - *true wenn inst (o) = c oder* $\exists$ *c1, c2, c3, ..., cn* $\in$ *NL mit inst(o) = c1* $\wedge$ *c1 isA c2* $\wedge$ *c2 isA c3* $\wedge$ *...* $\wedge$ *cn isA c*
  - *false sonst*

Zusätzlich zu ihrer Definition müssen objektorientierte Graphen weitere Bedingungen erfüllen. So müssen Attribute eines Objekts in der zum Objekt gehörenden Klasse definiert sein. Ebenso unterliegen Links aufgrund der zugehörigen Assoziation Bedingungen bezüglich der Kardinalität, des Typs, der Ordnungsnummer und der verbundenen Objekte. Diese Bedingungen werden in Definition 4.5 für wohlgeformte, objektorientierte Graphen zusammengefasst und beziehen sich im Wesentlichen auf die Ausprägung.

Soweit keine Angaben gemacht werden, werden in folgenden Definitionen tiefgestellte Indizes als Abkürzung für Zugriffe auf einzelne Teile eines Tupels verwendet. Zum Beispiel bezeichnet src_e und tgt_e den Ursprung und das Ziel eines Links $e \in E$, SI_G bezeichnet

die Schemainformation und Ext_G die Ausprägung eines objektorientierten Graphen *G=(SI,Ext)*.

Definition 4.5: Wohlgeformter objektorientierter Graph

Ein objektorientierter Graph *G := (SI, Ext)* mit *SI := (NL, EL, A, IsAs, Assocs, Attrs)* und *Ext := (N, E, inst, av)* wird wohlgeformt genannt, wenn für seine Ausprägung *Ext* bezogen auf seine Schemainformation *SI* gilt:

- $\forall a \in A$ und $\forall o \in N$ mit $av(o, a) = value$ gilt: Attributzuordnung
  $\exists c \in NL$ mit $instanceOf(o, c) = true \wedge$
  $Attrs(a) = (c, basetype) \wedge value \in basetype$
- $\forall el \in EL$ mit $Assocs(el) = (srcNI, srcCard, atype, abasetype, tgtNI, tgtCard)$ und
  - $qualified \in atype$ und $ordered \notin atype$ gilt: Qualifizierer
    - $\forall e1, e2 \in E$ mit $e1 \neq e2$ und $el_{e1} = el_{e2} = el$ und $src_{e1} = src_{e2}$ gilt:
      $q_{e1} \neq q_{e2} \wedge i_{e1} = i_{e2} = \{\varepsilon\} \wedge q_{e1} \in abasetype \wedge q_{e2} \in abasetype$
  - $ordered \in atype$ und $qualified \notin atype$ gilt: Ordnungsnummer
    - $\forall e1, e2 \in E$ mit $e1 \neq e2$ und $el_{e1} = el_{e2} = el$ und $src_{e1} = src_{e2}$
      gilt: $i_{e1} \neq i_{e2} \wedge q_{e1} = q_{e2} = \{\varepsilon\}$
  - $qualified \in atype$ und $ordered \in atype$ gilt:
    - $\forall e1, e2 \in E$ mit $e1 \neq e2$ und $el_{e1} = el_{e2}$ und $src_{e1} = src_{e2}$ und $q_{e1} = q_{e2}$
      gilt: $i_{e1} \neq i_{e2} \wedge q_{e1,} q_{e2} \in abasetype$
  - $srcCard = One$ gilt: Kardinalität
    - $\forall e1, e2 \in E$ mit $e1 \neq e2$ und $el_{e1} = el_{e2} = el$ gilt:
      $src_{e1} \neq src_{e2}$
  - $tgtCard = One$ gilt:
    - $\forall e1, e2 \in E$ mit $e1 \neq e2$ und $el_{e1} = el_{e2} = el$ gilt:
      $tgt_{e1} \neq tgt_{e2}$
  - $\forall e \in E$ mit $el_e = el$ gilt: Schemakonformität
    $instanceOf(src_e, srcNI) = true \wedge$
    $instanceOf(tgt_e, tgtNI) = true$

Die Bedingung Attributzuordnung schreibt vor, dass für Attributzuweisungen einer Ausprägung entsprechende Attribute in der zum Objekt gehörigen Klasse vorhanden sind. So darf ein Objekt kein Attribut enthalten, das nicht in der zugehörigen Klasse oder einer Vaterklasse definiert ist. Dies entspricht der Vererbung von Attributen in objektorientierten Sprachen.

Nachfolgende Bedingungen beziehen sich auf Links einer Assoziation. So müssen die Qualifizierer beziehungsweise die Ordnungsnummer eindeutig sein. Eine Ausnahme bilden qualifizierte, geordnete Assoziationen. Dabei können Links mit gleichem Qualifizierer auftreten, die aber aufgrund ihrer Ordnungsnummer eindeutig sein müssen. Ist eine

Assoziation entweder/weder qualifiziert oder/noch geordnet, so muss als Qualifizierer und/oder Ordnungsnummer *{ε}* verwendet werden.

Ebenso dürfen die Kardinalitäten einer Assoziation nicht verletzt werden. Definition 4.2 gemäß [Zün01] definiert nur obere Schranken für Kardinalitäten (*One* und *Many*). Die beiden ersten Bedingungen für Kardinalitäten sichern zu, dass keine zwei Links einer Assoziation mit Kardinalität *One* existieren, die das gleiche Objekt in der Rolle als *src* beziehungsweise *tgt* verwenden. Mehrere gleiche Links einer Assoziation zwischen zwei Objekten sind aufgrund der Definition als Kantenmenge nicht zulässig. Voraussetzung ist, dass für geordnete oder qualifizierte Assoziationen, die Ordnungsnummern beziehungsweise Qualifizierer alle Links der Assoziation eindeutig sind. Dies wird durch die Ordnungsnummerbedingung und Qualifiziererbedingung zugesichert.

Die letzte Bedingung sichert zu, dass die Objekte, zwischen denen ein Link existiert, schemakonform zu den Klassen der entsprechenden Assoziation sind. Durch diese Bedingungen ist ein wohlgeformter, objektorientierter Graph mit dem Klassen- und Instanzmodell in objektorientierten Sprachen wie zum Beispiel Java [Fla97] vergleichbar.

Die Menge alle Ausprägungen *Ext* eines Graphen *G=(SI, Ext)* bezogen auf seine Schemainformation *SI*, die die Wohlgeformtheitseigenschaften gemäß Definition 4.5 erfüllen, wird als *Graphklasse* bezeichnet. Die Menge wird als *GC(SI) = GraphClass (SI)* geschrieben.

4.2.5 Erweiterungen für Annotationen

Gefundene Entwurfsmusterinstanzen beziehungsweise Teilmusterinstanzen werden durch Annotationen am abstrakten Syntaxgraphen repräsentiert. In den vorherigen Abschnitten sind abstrakte Syntaxgraphen vorgestellt worden. Für einen annotierten abstrakten Syntaxgraphen ist es notwendig, die Schemainformation des abstrakten Syntaxgraphen aus den vorherigen Abschnitten entsprechend zu erweitern.

Als Erweiterung erhält die Schemainformation des abstrakten Syntaxgraphen einen eindeutigen Wurzelknoten. Alle Knoten der Schemainformation sind von diesem Wurzelknoten über Vererbungsbeziehungen direkt oder indirekt über weitere Knoten erreichbar. Dies ist keine Einschränkung der Allgemeinheit, da sich jedes Schema, das keinen Wurzelknoten besitzt, einfach durch Hinzufügen eines neuen Knotens und Vererbungsbeziehungen erweitern lässt.

Zum Beispiel existiert in Abbildung 4.5 kein Wurzelknoten, daher muss ein neuer Knoten mit Vererbungsbeziehungen zu ASGNode und ASGAttr hinzugefügt werden. In Abbildung 4.4 ist der ASGNode Knoten ein Wurzelknoten. Die Vererbungsbeziehungen sind dabei zur besseren Übersicht nur angedeutet.

Das Klassendiagramm in Abbildung 4.7 beschreibt die Erweiterung einer Schemainformation eines wohlgeformten, objektorientierten Graphen, der einen eindeutigen Wurzelknoten besitzt.

Die Klasse ASGNode in Abbildung 4.7 ist der Wurzelknoten der Schemainformation des abstrakten Syntaxgraphen. Damit der abstrakte Syntaxgraph annotiert werden kann, werden zwei neue Knoten AbstractNode und Annotation hinzugefügt. Die Namen der Knoten müssen dabei eindeutig in der erweiterten Schemainformation sein.

Annotationen des abstrakten Syntaxgraphen werden durch Objekte der Klasse Annotation repräsentiert. Eine Annotation besitzt einen Bezeichner, repräsentiert durch das kind Attribut, und kann beliebige weitere Attribute besitzen. Für die weiteren Definitionen wird das Attribut value mit Typ Float vorausgesetzt, dass rationale Zahlen speichert.

Durch die qualifizierte annos-Assoziation zwischen Annotation und AbstractNode können Annotationen mit Elementen des abstrakten Syntaxgraphen sowie anderen Annotationen verbunden werden. Der Qualifizierer muss dabei eindeutig sein.

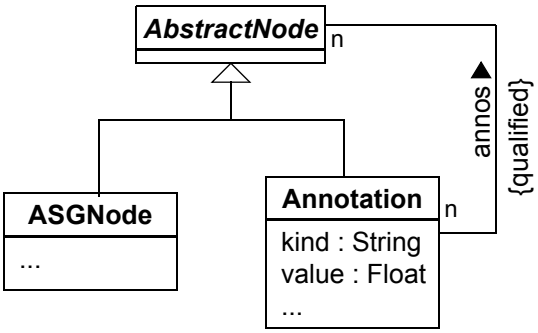


Abbildung 4.7 Annotationserweiterung für abstrakten Syntaxgraphen

Die Erweiterung einer Schemainformation gemäß Abbildung 4.7 unterliegt geringen Beschränkungen - die Schemainformation des zu annotierenden Graphen besitzt einen Wurzelknoten - was eine einfache technische Umsetzung ermöglicht. In der Praxis kommen allerdings oftmals Verfeinerungen zum Einsatz. So werden zum Beispiel verschiedene Annotationen nicht aufgrund ihres Bezeichners - kind Attributs - unterschieden, sondern es wird für jede Annotation eine eigene Klasse erzeugt. Die Klasse Annotation ist dann ein Wurzelknoten aller Annotationen. Weiterhin besteht die Möglichkeit der Verfeinerung der qualifizierten annos-Assoziation. Dabei wird die annos-Assoziation durch einzelne Assoziationen zwischen einzelnen Annotationsklassen ersetzt.

Eine gemäß Abbildung 4.7 erweitere Schemainformation erfüllt die Eigenschaften eines wohlgeformten, objektorientierten Graphen und wird im Folgenden als Grundlage für die Definition von Musterinstanzregeln in Regelkatalogen verwendet.

Alternativ können Musterinstanzregeln auch auf Basis anderer graphischer Repräsentationen des Quelltextes wie zum Beispiel Kontrollfluss- oder Datenflussgraphen definiert werden. Die verwendete Schemainformation muss allerdings durch die in Abbildung 4.7 vorgestellte Annotationserweiterung ergänzt werden können.

4.3 Regelkataloge

Entwurfsmusterinstanzen werden durch Musterinstanzregeln auf Basis eines objektorientierten, annotierbaren abstrakten Syntaxgraphen definiert. Dabei ist die Schemainformation des abstrakten Syntaxgraphen gemäß Abbildung 4.7 erweitert worden.

Definition 4.6: Musterinstanzregel

Eine Musterinstanzregel ist definiert als *rule := (RL, AL, SI, SG, an, bf, th)* mit

- *RL* *Name der Regel*
- *AL* *Menge von Annotationsbezeichnern*
- *SG = (N, E, inst, av)* $\in$ *GC(SI) und ist zusammenhängend*
- *an* $\subseteq AL \times N$ Annotationsrelation
- *bf* $\in [0..1]$ Belief
- *th* $\in [0..1]$ Threshold

Jede Musterinstanzregel besteht aus einem Namen *RL* und einem Graphen *SG* bezogen auf eine Schemainformation *SI*. Durch die Annotationsrelation *an* werden Knoten des Graphen *SG* Annotationsbezeichnungen *AL* zugewiesen. Die Annotationsrelation beschreibt, welche Teile des Graphen *SG* bei einer erfolgreichen Ausführung der Musterinstanzregel annotiert werden.

Zu einer Musterinstanzregel gehören ein Genauigkeitswert (Belief) *bf* und ein Schwellwert (Threshold) *th.* Der Genauigkeitswert und der Schwellwert einer Regel wird in Abschnitt 4.4.1 erläutert.

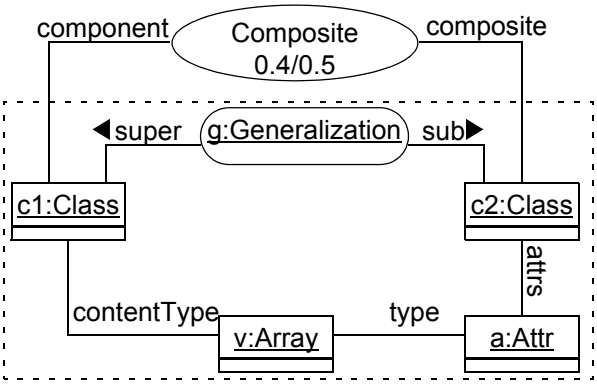


Abbildung 4.8 Composite-Regel des Beispielkatalogs

Abbildung 4.8 zeigt die Composite-Regel des Beispielkatalogs aus Abbildung 4.1. Der RegelASGausschnitt *SG* ist durch die gestrichelte Linie gekennzeichnet. Der Name der Musterinstanzregel *RL* sowie der Genauigkeitswert *bf* und Schwellwert *th* werden durch den Musterinstanzknoten Composite dargestellt. Die Annotationsbezeichner sind component und composite, und die Annotationsrelation wird durch Linien vom Musterinstanz-

knoten Composite zu den beiden Knoten c1:Class und c2:Class des RegelASGausschnitts *SG* dargestellt.[1]

Eine Menge von Musterinstanzregeln bilden einen Regelkatalog, bei dem zusätzlich Verfeinerungsbeziehungen zwischen Musterinstanzregeln definiert werden können.

Definition 4.7: Regelkatalog

Ein Regelkatalog *RK := (rSI, R, rf)* besteht aus:

- *rSI* *verfeinerte Schemainformation gemäß Definition 4.8*
- *R* *ist eine Menge von Musterinstanzregeln*
- *rf:* $R \rightarrow \{R \cup \{\perp\}\}$ *ist total und zyklenfrei* Verfeinerungsfunktion

Die verfeinerte Schemainformation *rSI* dient als Basis für die im Regelkatalog enthaltenen Musterinstanzregeln *rule = (RL, AL, rSI, SG, an, bf, th)*. Die Konstruktion der verfeinerten Schemainformation für einen Regelkatalog wird in Definition 4.8 beschrieben. Die Verfeinerungsfunktion *rf* ist total und zyklenfrei. Durch die Totalität ist keine Mehrfachverfeinerung möglich. Für die Darstellung der Musterinstanzregeln eines Katalogs sowie der Verfeinerungsfunktion werden Klassendiagramme verwendet und als *Regeldefinitionsdiagramm* bezeichnet.

Die Basis eines Regelkatalogs beziehungsweise der in ihm vorhandenen Musterinstanzregeln besteht aus der verfeinerten Schemainformation eines wohlgeformten, objektorientierten, annotierten abstrakten Syntaxgraphen. Die verfeinerte Schemainformation *rSI* wird aufgrund der Regelnamen und der Verfeinerungsrelation *rf* aus der erweiterten Schemainformation gemäß Abbildung 4.7 konstruiert.

Definition 4.8: Verfeinerte Schemainformation

Eine verfeinerte Schemainformation $rSI := (NL_{rSI}, EL_{rSI}, A_{rSI}, IsAs_{rSI}, Assocs_{rSI}, Attrs_{rSI})$ wird aus einem Regelkatalog *RK = (rSI, R, rf)* und der Schemainformation eines wohlgeformten, objektorientierten, annotierbaren abstrakten Syntaxgraphen *SI := (NL, EL, A, IsAs, Assocs, Attrs)* wie folgt konstruiert:

- $NL_{rSI} = NL \cup_{\text{rule} \in R} (RL)$
- $EL_{rSI} = EL$
- $A_{rSI} = A$
- $IsAs_{rSI}$: $NL_{rSI} \rightarrow NL_{rSI}$ *mit*
  $n1\ isA_{rSI}\ n2: \forall n1, n2 \in NL$ *und* $n1\ isA\ n2$
  $r1\ isA_{rSI}\ r2: \forall r1, r2 \in R$ *mit* $rf(r1) = r2$
  $r1\ isA_{rSI}\ Annotation \in NL: \forall r \in R$ *mit* $rf(r) = \varepsilon$
- $Assocs_{rSI} = Assocs$
- $Attrs_{rSI} = Attrs$

1. Die unterschiedliche Darstellung der Musterinstanzknoten, Knoten des ASG und Knoten, die von anderen Musterinstanzregeln erzeugt worden sind, dient lediglich der besseren Lesbarkeit.

Dazu wird für jede Musterinstanzregel (jeden Regelnamen) ein Knoten in *rSI* erzeugt. Für die Verfeinerungsfunktion *rf* wird entsprechend die Verfeinerungsfunktion *IsAs* der Schemainformation erweitert. Für den Fall, dass ein Regelname auf $\perp$ abgebildet wird, wird die *IsAs* Relation auf den Annotationsknoten abgebildet, vgl. Abbildung 4.9. Durch diese Konstruktion gilt $\forall r \in RL : instanceOf\ (r, Annotation) = true$.

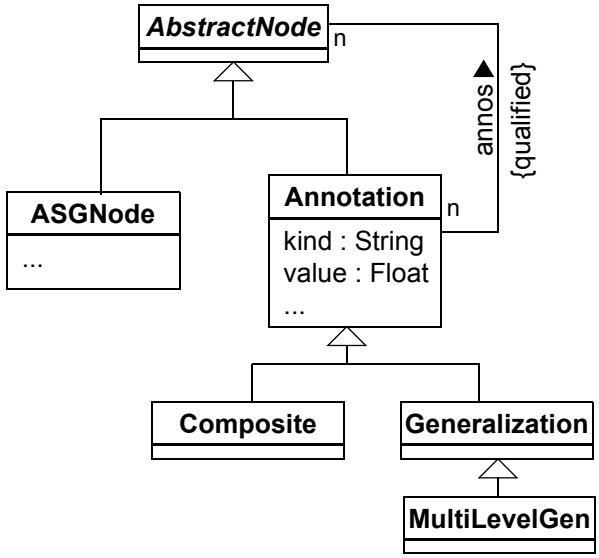


Abbildung 4.9 Verfeinerte Schemainformation für Beispielkatalog

Im Klassendiagramm von *rSI* existiert also für jede Musterinstanzregel eine Klasse, und die Klassen sind durch Vererbungsbeziehungen verbunden. Abbildung 4.9 zeigt einen Ausschnitt der verfeinerten Schemainformation des Beispielkataloges.

Der Beispielregelkatalog in Abbildung 4.1 enthält neben der Composite-Regel noch eine Generalization-Regel und eine MultiLevelGen-Regel. Diese drei Regeln sind dabei voneinander abhängig.

Definition 4.9: Regelabhängigkeit

Gegeben sind zwei Musterinstanzregeln $r1 = (RL1, AL1, rSI, SG1, an1, bf1, th1) \in R$ und $r2 = (RL2, AL2, rSI, SG2, an2, bf2, th2) \in R$ eines Regelkatalogs $RK := (rSI, R, rf)$. Dann ist die Funktion $dep : R \times R \rightarrow boolean$ definiert als:

$dep\ (r1, r2) = true$ wenn $\exists n \in N_{SG1}$ und $inst\ (n) = RL2$
sonst *false*

Zwei Musterinstanzregeln sind abhängig, wenn im RegelASGausschnitt der ersten Musterinstanzregel eine Annotation der zweiten Musterinstanzregel verwendet wird. Die Regelabhängigkeit beschreibt damit gleichzeitig die Kompositionsbeziehungen zwischen den Musterinstanzregeln. Zum Beispiel ist im Beispielregelkatalog die Composite-Regel von der Generalization-Regel, nicht aber von der MultiLevelGen-Regel abhängig.

Definition 4.10: Konsistenter Regelkatalog

Ein Regelkatalog *RK := (rSI, R, rf)* ist konsistent, wenn

(allgemeine Bedingungen)

- $RLk := \cup_{rule \in R} RL$ *und es gilt:* $|RLk| = \Sigma_{rule} |RL|$
- $ALk := \cup_{rule \in R} AL$ *und es gilt:* $|ALk| = \Sigma_{rule} |AL|$

(Regelbedingungen)

- $\neg\exists$ *r1, r2, ..., rn* $\in R$ *mit dep(r1, r2) = true* $\wedge$ *...* $\wedge$ *dep(rn, r1) = true*
- $\forall$*r1, r2, ..., rn-1, rn* $\in R$ *mit dep(r1, r2) = true* $\wedge$ *...* $\wedge$ *dep(rn-1, rn) = true gilt:*
  $\neg\exists$ *s2, ..., sm* $\in R$ *mit m > 0 und rf (rn) = sm* $\wedge$ *...* $\wedge$ *rf(s2) = r1*
- $\forall$*r1, rn* $\in R$ *mit rf(rn) = r1 gilt:*
  $\neg\exists$*s2, ..., sn-1* $\in R$ *mit dep(r1,s2)=true* $\wedge$ *...* $\wedge$ *dep(sn-1,rn)=true*
- $\forall$*r1, rn* $\in R$ *mit rf(rn) = r1 und* $\exists$*s2, ..., sn-1* $\in R$ *mit*
  dep(rn, sn-1) = true $\wedge$ *...* $\wedge$ *dep(s2, r1) = true gilt:*
  $|Nm| > 1$ *mit* $Nm = \{n \in N_{rn} | nl_{rn}(n) = nnr1$ *und rf(nnr1) = r1}*

(Qualifiziererbedingungen)

- *für alle Musterinstanzregeln r1, r2* $\in R$ *und rf (r2) = r1 gilt:*
  $\forall a \in ALk$ *mit* $(a, n1) \in an_{r1}$ *und* $n1 \in N_{SGr1}$ *gilt:* $(a, n2) \in an_{r2} \wedge n2 \in N_{SGr2}$
- $\forall r1 \in R$ *mit* $nr1 \in N_{SGr1}$ *und* $nnr1 := nl_{SGr1}(nr1) \in RLk$ *und* $\forall q \in ALk$ *und*
  $\forall nntgt := nl_{SGr1}(tgt)$ NL_{SIe} *mit el := (nr1, annos, i, q, tgt)* $\in RLk$ *gilt*
  $\exists r2 := ru(nnr1) \in R$ *und* $\exists nr2 \in N_{SGr2}$ *mit* $(nr2, q) \in an_{r2}$ *und*
  $\exists nnr2 := nlr2(nr2)$ *mit instanceOf (nr1, nnr2) = true*

Um einen Regelkatalog für die automatische Erkennung von Entwurfsmusterinstanzen einsetzen zu können, muss er eine Reihe von Konsistenzbedingungen einhalten. Die Konsistenzbedingungen ergeben sich dabei in erster Linie aus den Regelabhängigkeiten und der Verfeinerungsfunktion eines Regelkatalogs.

Die Konsistenzbedingungen für einen Regelkatalog werden in drei Gruppen eingeteilt. Allgemeine Bedingungen sichern die Eindeutigkeit der Regel- und Annotationsbezeichner zu, Regelbedingungen beschreiben Bedingungen bezüglich der Abhängigkeit und Verfeinerung der Musterinstanzregeln. Qualifiziererbedingungen sind Bedingungen für eine konsistente Verwendung der Annotationsbezeichner im Regelkatalog. Die beschriebenen Konsistenzbedingungen entsprechen dabei den informal beschriebenen Bedingungen aus 4.1.

Die erste Regelbedingung besagt, dass Musterinstanzregeln nicht zyklisch abhängig sein dürfen. Die Musterinstanzregel r1 hängt gemäß Definition 4.9 von Musterinstanzregel r2 ab, wenn der Graph der Musterinstanzregel r1 Knoten besitzt, dessen zugehöriger Knoten in der Schemainformation aufgrund der Musterinstanzregel r2 erzeugt worden ist. Der Beispielregelkatalog in Abbildung 4.1 ist zyklenfrei.

Regelbedingung zwei verlangt, dass zwei Musterinstanzregeln, die gegebenenfalls über weitere Musterinstanzregeln voneinander abhängig sind, nicht gleichzeitig auch über mehrere Musterinstanzregeln verfeinert werden dürfen. Zusätzlich verlangt die dritte Regelbedingung, dass eine Musterinstanzregel eine andere Musterinstanzregel verfeinert, die von ihr abhängig ist. Dadurch wird sichergestellt, dass durch die Polymorphieeigenschaft des Graphmodells während der Regelausführung keine Zyklen entstehen. Diese Bedingung ist im Beispielregelkatalog erfüllt.

Zwei Musterinstanzregeln, die (über mehrere Musterinstanzregeln) voneinander abhängig sind und sich gleichzeitig direkt verfeinern, werden als *quasirekursiv* bezeichnet. Für solche Musterinstanzregeln verlangt die vierte Regelbedingung, dass die Anzahl der Knoten der abhängigen Musterinstanzregel in der verfeinernden Musterinstanzregel mindestens größer ist als 1. Durch diese Bedingung wird sichergestellt, dass die spätere Regelausführung terminiert, weil sich die Anzahl möglicher Anwendungsstellen um mindestens 1 verringert.

Die erste Qualifiziererbedingung verlangt, dass eine Musterinstanzregel r2, die eine Musterinstanzregel r1 verfeinert, mindestens allen Annotationsbezeichnern, die in der Musterinstanzregel r1 verwendet werden, ebenfalls einen entsprechenden Knoten zuordnet. Es sei dazu bemerkt, dass eine Musterinstanzregel r2, die eine Musterinstanzregel r1 verfeinert, keine semantische Verfeinerung sein muss, sondern lediglich diese "strukturelle" Bedingung erfüllen muss. Im Beispielregelkatalog in Abbildung 4.1 erfüllt die MultiLevelGen-Regel diese Bedingung, da die sub und super Annotationsbezeichner, die in der Generalization-Musterinstanzregel vorkommen, ebenfalls vorhanden sind. Das zusätzlich noch ein upper Annotationsbezeichner verwendet wird, verletzt diese Bedingung nicht.

Die zweite Qualifiziererbedingung beinhaltet zwei verschiedene Bedingungen. Die erste Bedingung verlangt, dass Qualifizierer, die in einem Graphen einer Musterinstanzregel verwendet werden, in der Annotationsrelation der entsprechenden Musterinstanzregel verwendet werden. Die zweite Bedingung verlangt, dass die annotierten Zielknoten im Graphen einer Musterinstanzregel und die Knoten in der Annotationsrelation der entsprechenden Musterinstanzregel, bezogen auf die verfeinerte Schemainformation, konform sind. Für den Beispielregelkatalog aus Abbildung 4.1 ist auch diese Bedingung erfüllt. Die Bedingung wäre zum Beispiel verletzt, wenn einer der Qualifizierer sub oder super in der Composite-Regel einen anderen Namen hätte oder einer der annotierten Knoten zum Beispiel v:Array oder a:Attr wäre.

Die Qualifiziererbedingungen könnten auch durch eine Verfeinerung der qualifizierten annos Beziehung ersetzt werden. Die Verfeinerungen würden ähnlich der Verfeinerung der Annotationen aufgrund der Musterinstanzregeln konstruiert werden. Ein solches Vorgehen ist theoretisch ohne größere Probleme realisierbar, allerdings entstünden dann in der verfeinerten Schemainformation Assoziationen zwischen Annotationsknoten und Knoten des abstrakten Syntaxgraphen. Dies würde zu einer Änderung des Schemas des abstrakten Syntaxgraphen führen, sobald neue Musterinstanzregeln zum Katalog hinzu-

gefügt werden oder Annotationskanten sich ändern. Durch den interaktiven Analyseprozess entstehen so häufig Schemainformationsänderungen. Bei automatisch generierten Schemainformationen, zum Beispiel aus einer Grammatik, ist die hier beschriebene Definition eines Regelkatalogs in der technischen Umsetzung leichter realisierbar.

4.4 Regelkatalogsemantik

Die Semantik eines Regelkatalogs wird durch die Abbildung eines Regelkatalogs auf das in [Zün01] vorgestellte formale Regelsystem definiert. Graphtransformationsregeln werden im formalen Regelsystem als so genannte *Story-Pattern* bezeichnet. Ein Story-Pattern *SG := (LG, RG)* besteht aus zwei Graphen *LG, RG* $\in$ *GC(SI)*, die die linke und rechte Regelseite einer Graphtransformationsregel darstellen. Die beiden Graphen müssen wohlgeformte, objektorientierte Graphen sein und sich auf die gleiche Schemainformation beziehen.

Die Ausführung eines Story-Patterns auf einen Graphen besteht dabei aus der Suche einer Anwendungsstelle für die Regel, die durch die linke Regelseite definiert ist, und der anschließenden Ersetzung der Anwendungsstelle durch die rechte Regelseite. Die Wahl der Anwendungsstelle ist dabei nicht-deterministisch, was zu einer Menge möglicher Ergebnisgraphen führt.

Als Notation für Story-Pattern werden UML-Kollaborationsdiagramme verwendet, wobei die linke und rechte Regelseite in einem Diagramm dargestellt werden. Hinzuzufügende Elemente werden mit «create» und zu löschende Elemente mit «destroy» gekennzeichnet. Die Grenzen der Regel sind durch einen Rahmen festgelegt. Weitere Sprachkonstrukte sind zum Beispiel boolesche Bedingungen, negative Objekte und Links, optionale Objekte und Multiobjekte.

Story-Pattern können durch Kontrollflüsse miteinander verknüpft werden. Diese Kontrollflüsse werden als Methoden von Klassen definiert und beschreiben gleichzeitig einen Namensraum. Methodenaufrufe und Parameterübergaben werden - ähnlich zu Programmiersprachen - über einen globalen Stack realisiert.

Die Semantik eines Regelkatalogs *RK = (rSI, R, rf)* ist durch eine Abbildung der Musterinstanzregeln *R* auf eine Menge von Story-Pattern definiert, wobei jeder Musterinstanzregel ein Story-Pattern zugeordnet wird. Die verfeinerte Schemainformation rSI konstruiert gemäß Definition 4.8 ist dabei die Grundlage für die linke und rechte Regelseite der Story-Pattern. Der zu analysierende Quelltext liegt als abstrakter Syntaxgraph ebenfalls bezüglich der verfeinerten Schemainformation *rSI* vor.

Die Story-Pattern, die für jede Musterinstanzregel eines Regelkatalogs konstruiert werden, sind im Wesentlichen durch den RegelASGausschnitt einer Musterinstanzregel beschrieben. Die linke Regelseite *LG* ist mit dem RegelASGausschnitt *SG* identisch. Die rechte Regelseite *RG* beinhaltet *SG*, und zusätzlich wird der rechten Regelseite ein Objekt, das dem Musterinstanzknoten entspricht, hinzugefügt. Das Objekt wird als *Muster-*

instanzobjekt bezeichnet. Das Objekt ist dabei Instanz der zugehörigen Klasse der Musterinstanzregel in der Schemainformation, die aufgrund der Konstruktion der Schemainformation in Definition 4.8 erzeugt worden ist.

Definition 4.11: Abbildung von Musterinstanzregeln auf Story-Pattern

Für eine Musterinstanzregel *rule = (RL, AL, rSI, SG, an, bf, th)* eines Regelkatalogs *RK = (rSI, R, rf)* mit *rule* $\in$ *R* wird ein Story-Pattern *SP = (LG, RG)* mit *LG, RG* $\in$ *GC(rSI)* und booleschen Bedingungen *BC* folgendermaßen konstruiert:

- *LG = SG*
- *RG = (N, E, inst, av) mit*
  - $N_{RG} = N_{SG} \cup RL$
  - $E_{RG} = E_{SG} \cup AL$
  - $inst_{RG}(n) = inst_{SG}(n)\ \forall n \in N_{SG}$ *und* $inst_{RG}(RL) = RL$
  - $av_{RG}(n, a) = av_{SG}(n, a)$
    $\forall n \in N_{SG}, \forall a \in AL$ *und instanceOf (n, Annotation) = false*
    $av_{RG}(n, value) = th\ \forall n \in N_{SG}$ *und instanceOf (n, Annotation) = true*
    av_{RG} *(rulename, value) = min (bf, {v| v :=* av_{SG} *(n, value) mit* $n \in N_{SG}$ *und instanceOf (n, Annotation) = true})*
- $\forall n \in N_{LG}$ *und instanceOf (*$inst_{LG}$ *(n), Annotation) = true wird folgende boolesche Bedingung BC als OCL-Ausdruck erzeugt:*
  col := AND [e = (n, annos, i, q,nx) $\in E_{LG}$*] (nx.ref (q))* $\wedge$
  col.iterate (x:Float, m : Float = 0 | m = (m >= x) ? m : x.value) $\wedge$
  n.value = m

Die Annotationsrelation wird auf entsprechende Links in der rechten Regelseite abgebildet. Bei der Ausführung wird also, wenn der RegelASGausschnitt gefunden worden ist, ein neues Objekt - eine neue Annotation - erzeugt und entsprechende Annotationslinks zu einzelnen Teilen der Anwendungsstelle gezogen. Abbildung 4.10 zeigt das Story-Pattern der Composite-Regel des Beispielkatalogs aus Abbildung 4.1.

Der Schwellwert einer Musterinstanzregel wird zu einer Attributbedingung aller Annotationsobjekte der linken Regelseite, die Instanzen der Klasse *Annotation* sind. Dies beschränkt die Anwendungsstellen auf die, deren Präzisionswert *value* größer ist als der Schwellwert. Die Beschränkung verhindert, dass zu unpräzise Teilergebnisse, Annotationen mit niedrigem Präzisionswert, nicht weiter für darauf aufbauende Musterinstanzregeln verwendet werden. Zum Beispiel führen nur Generalization-Annotationen mit einem Präzisionswert größer oder gleich *0.5* zu einer Composite-Annotation (value >= 0.5).

Die booleschen Bedingungen, die in Form von OCL-Ausdrücken [UML] erzeugt werden, beschränken, ebenso wie Attributbedingungen der Schwellwerte, die möglichen Anwendungsstellen. Das *AND* bildet die Schnittmenge aller Kollektionen der Objekte, die über Rückwärtsnavigation *ref* eines Annotationslinks erreichbar sind. Das Maximum wird

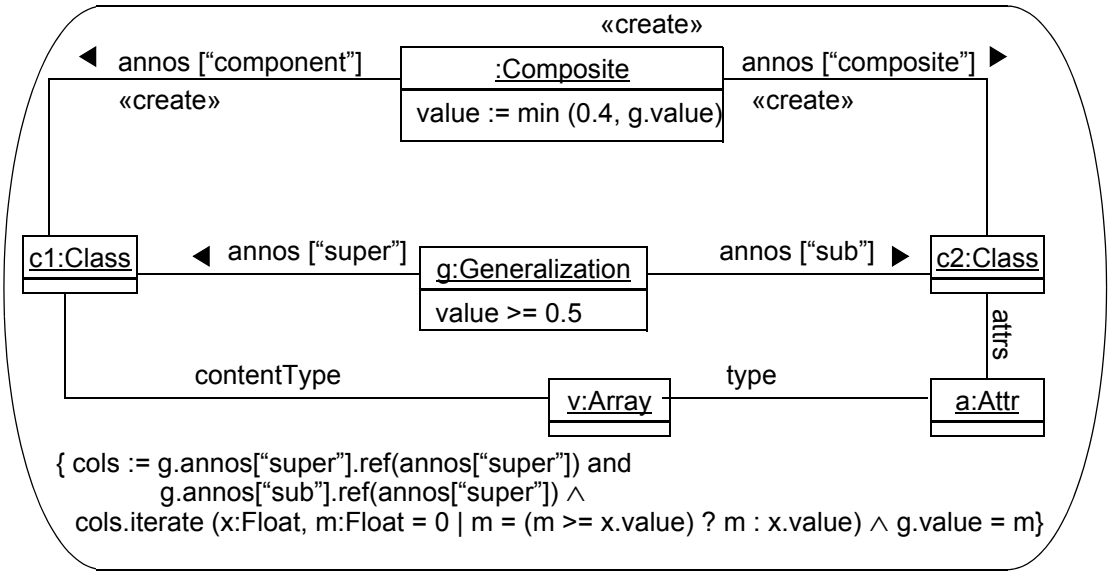


Abbildung 4.10 Story-Pattern für Composite-Regel

durch die *iterate* Operation auf der resultierenden Kollektion bestimmt. Die Menge der möglichen Anwendungsstellen wird also auf diejenigen begrenzt, die maximale Präzisionswerte besitzen. Würden zum Beispiel bei der Ausführung der Composite-Regel aus Abbildung 4.1 zwei Generalization-Annotationen *g1* und *g2* zwischen den beiden Klassen c1:Class und c2:Class mit unterschiedlichen Präzisionswerten ($value_{g1} < value_{g2}$) existieren, dann wird in diesem Fall die Generalization-Annotation *g2* aufgrund des höheren Präzisionswerts gewählt.

Der Präzisionswert einer neuen Annotation *value* wird dabei berechnet als das Minimum aller Präzisionswerte von Annotationen der Anwendungsstelle und des Genauigkeitswerts *bf* der Musterinstanzregel (value := min (0.4, g.value)).

Bei der Semantikdefinition von Regelkatalogen durch Konstruktion der Schemainformation und die Abbildung der Musterinstanzregeln auf Story-Pattern muss gezeigt werden, dass die Anwendung eines Story-Patterns auf einem wohlgeformten, objektorientierten Graphen als Ergebnis wiederum einen wohlgeformten, objektorientierten Graphen ergibt.

Für Story-Pattern, wie sie in [Zün01] vorgestellt werden, ist im Allgemeinen ein solcher Beweis sehr aufwändig, was an den zum Teil sehr komplexen Graphtransformationen liegt. Für die hier verwendeten stark eingeschränkten Story-Pattern die gemäß Definition 4.11 konstruiert worden sind ist dies einfach nachzuweisen.

Story-Pattern, die aus Musterinstanzregeln eines Regelkatalogs erzeugt worden sind, enthalten nur Graphtransformationen, die ein Objekt und eine Menge von Links hinzufügen. Das Objekt ist aufgrund der Konstruktion der Schemainformation immer Instanz einer Subklasse der Klasse Annotation.

Die Annotationslinks beziehen sich auf die annos-Assoziation in Abbildung 4.7, und die annotierten Objekte sind Teil des wohlgeformten, objektorientierten Graphen, auf dem das Story-Pattern ausgeführt worden ist. Dem Präzisionswert des Annotationsobjekts entspricht das value Attribut der Klasse Annotation. Das Ergebnis der Ausführung der konstruierten Story-Pattern ist also ein wohlgeformter, objektorientierter Graph.

```
1: class A {}
2: class B inherits A {}
3: class C inherits B
4: { A[] children;}
```

Abbildung 4.11 Quelltextbeispiel mit mehrstufiger Vererbung

Zur Illustration der Anwendung der konstruierten Story-Pattern auf einen abstrakten Syntaxgraphen soll der Beispielquelltext in Abbildung 4.11 dienen. Die Musterinstanzregeln des Regelkatalogs aus Abbildung 4.1 sind in Story-Pattern gemäß Definition 4.11 überführt worden.

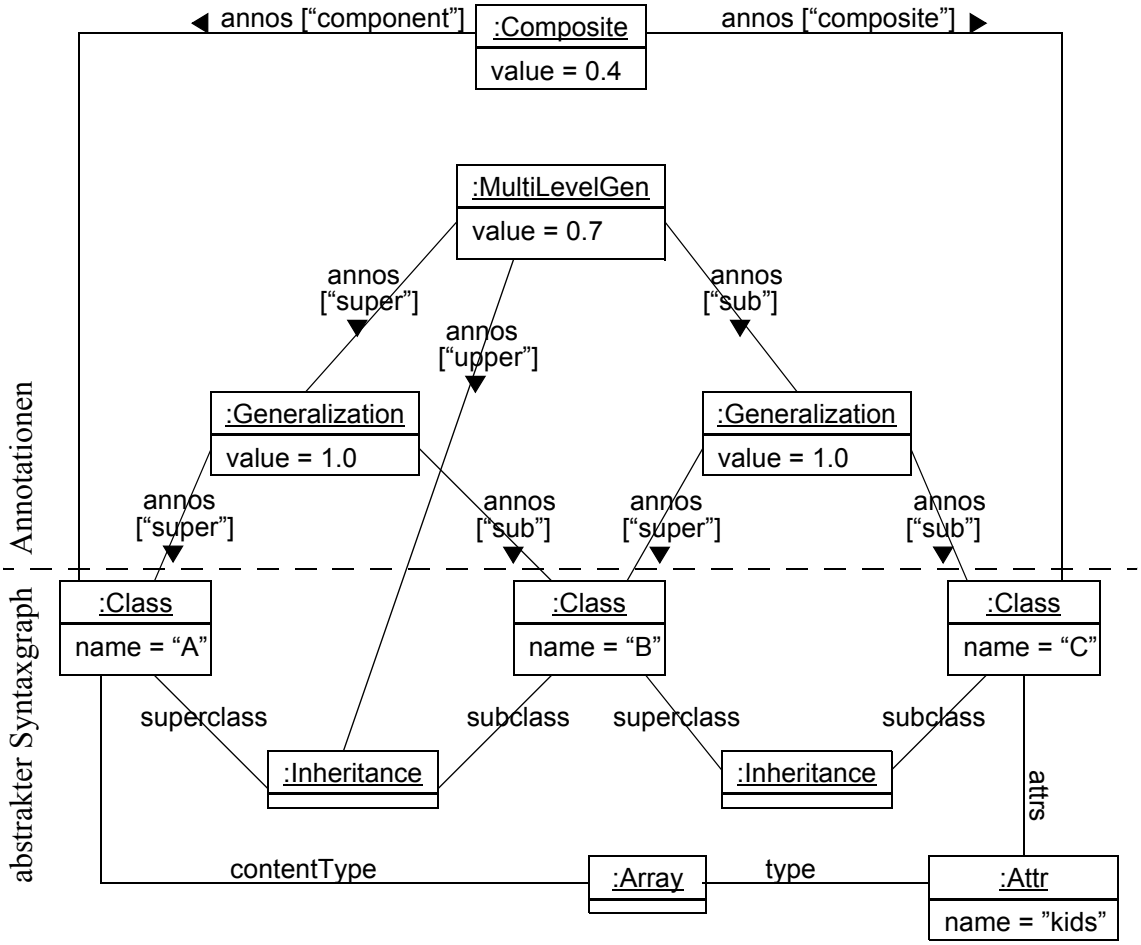


Abbildung 4.12 Story-Pattern Anwendungsbeispiel

Das Ergebnis ist in Abbildung 4.12 dargestellt und beinhaltet in der unteren Hälfte den abstrakten Syntaxgraphen des oben angegebenen Quelltextes. Die Schemainformation des abstrakten Syntaxgraphen ist dabei nicht dargestellt, ist aber der aus Abbildung 4.4 ähnlich. In der oberen Hälfte sind die Annotationsobjekte dargestellt, die von den entsprechenden Story-Pattern erzeugt worden sind. Das Story-Pattern der Generalization-Regel - oder kurz die Generalization-Regel - ist dabei zweimal angewendet worden und die MultiLevelGen-Regel und Composite-Regel sind jeweils einmal angewendet worden.

Das Composite-Annotationsobjekt :Composite - oder kurz Compositeannotation - markiert durch die Annotationskanten “composite” und “component” diejenigen Klassen, die die Rolle der Component-Klasse und die Rolle der Composite-Klasse des Composite-Gamma-Musters annehmen. In diesem Fall die Klasse A und Klasse C, repräsentiert durch entsprechende Objekte im abstrakten Syntaxgraphen. Die Generalization-Regel ist dabei aufgrund der MultiLevelGen-Regel angewendet worden. Die gefundene Composite-Entwurfsmusterinstanz hat dabei einen Präzisionswert von 0.4, was dem Genauigkeitswert der Regel entspricht.

Im Beispiel von Abbildung 4.12 sind alle Annotationen erzeugt worden und die Anwendung der Story-Pattern kann beendet werden. Für die Terminierung der Anwendung einer Menge von Story-Pattern darf erstens ein Story-Pattern an jeder Anwendungsstelle nur einmal positiv angewendet werden. Andernfalls würden immer neue gleiche Annotationen an derselben Anwendungsstelle erzeugt. Zweitens darf beim Fehlschlagen der Anwendung eines Story-Pattern dies erst dann wieder angewendet werden, wenn alle anderen Story-Pattern versucht worden sind anzuwenden. Sollte keines der anderen Story-Pattern mehr angewendet werden können, so kann die Anwendung beendet werden. Beide Bedingungen sind nicht Bestandteil der Definition von Story-Pattern in einer Implementierung allerdings sinnvoll und einfach umzusetzen. Die beiden Bedingungen beinhalten allerdings nicht, dass Zyklen aufgrund von erzeugten Objekten entstehen können. Bei Story-Pattern, die gemäß Definition 4.11 erzeugten worden sind, entstehen solche Zyklen allerdings nicht.

Dadurch, dass Musterinstanzregeln nicht zyklisch voneinander abhängig sein können (2. Regelbedingung aus Definition 4.10), besteht lediglich das Problem, dass aufgrund der Verfeinerung der Musterinstanzregeln eine Quasiabhängigkeit entsteht. Dies betrifft die quasirekursiven Musterinstanzregeln, wie zum Beispiel die Generalization- und MultiLevelGen-Regel im Beispielkatalog in Abbildung 4.1. Die 3. Regelbedingung in Definition 4.10 verlangt bei einer solchen Abhängigkeit, dass der RegelASGausschnitt mindestens zwei Knoten enthält, die durch andere Musterinstanzregeln erzeugte Annotationen repräsentieren. Dadurch wird die Anzahl der Anwendungsstellen um mindestens 1 reduziert. Das heißt, dass die Anzahl der Ausführungen eines Story-Patterns begrenzt ist, die Ausführung aller Story-Pattern und damit die automatische Analyse terminiert.

4.4.1 Fuzzygrammatiken

Der Genauigkeitswert einer Musterinstanzregel sowie die Minimum- und Maximumberechnungen in Story-Pattern für eine Musterinstanzregel gemäß Definition 4.11 beruhen auf der Theorie Fuzzygrammatiken [Zad65, Zad78]. Fuzzygrammatiken verknüpfen Grammatiken mit Fuzzymengen.

Eine Fuzzygrammatik ist ein Sechs-Tupel $FG := (VT, VN, P, S, J, f)$, wobei VT, VN endliche Mengen von Terminalen beziehungsweise Nicht-Terminalen über einem Alphabet sind. P ist die Menge der Produktionsregeln, S ist das Startsymbol. Diese Teile entsprechen denen einer "normalen" Grammatik. Der Fuzzyanteil ist die Menge J von Regelnamen und die Funktion $f : J \rightarrow [0, 1]$, die der Zugehörigkeitsfunktion in Fuzzymengen entspricht. Dadurch wird jeder Regel ein so genannter *Fuzzybelief* zugeordnet. Für ein Element $x \in L(FG)$ der Sprache, die durch die Grammatik gebildet wird, berechnet sich der Grad der Zugehörigkeit als das Maximum aller möglichen Ableitungen von x ausgehend vom Startsymbol S. Die Zugehörigkeit für eine Ableitung von x ausgehend von S ist dabei das Minimum aller Fuzzybeliefs der Regeln, die für die Ableitung verwendet werden. Der Fall $\forall r \in J$ gilt $f(r) = 1$ entspricht einer "normalen" Grammatik.

Diese Definition der Fuzzygrammatiken ist für Regelkataloge analog übernommen worden. Die Menge aller Regelnamen $RL_k = \cup_{\text{rule}} RL$ eines Regelkatalogs $RK = (rSI, R, rf)$ entspricht der Menge J von eindeutigen Namen. Zusammen mit den Genauigkeitswerten bf der Musterinstanzregeln $rule= (RL, AL, rSI, SG, an, bf, th)$ eines Regelkatalogs kann die Zugehörigkeitsfunktion $f : J \rightarrow [0, 1]$ bestimmt werden.

Die Minimumbildung der neu hinzuzufügenden Objekte in den Story-Pattern entspricht der Zugehörigkeit eines Elements (Quelltextausschnitts) bezogen auf eine Ableitung (Story-Patternanwendung). Dabei wird das Minimum nur aus den Annotationsobjekten berechnet, weil implizit die Objekte des abstrakten Syntaxgraphen den Wert 1 besitzen und dies keine Auswirkungen auf die Berechnung hat.

Die Maximumbildung in den booleschen Bedingungen sorgt dafür, dass die Zugehörigkeit eines Elements (Quelltextausschnitts) das Maximum aller Ableitungen ist. Auch hier werden nur die Werte der Annotationsobjekte berücksichtigt, weil Objekte des abstrakten Syntaxgraphen keine Ableitung darstellen. Der Wert wird bei der Ausführung eines Story-Patterns berechnet und dem neuen Annotationsobjekt zugewiesen und ist der Präzisionswert der Annotation.

Der Schwellwert, der als Attributbedingung in den Annotationsobjekten verwendet wird, hat keine direkte Entsprechung in Fuzzygrammatiken. Er entspricht dem Begrenzungswert einer λ-Fuzzy-Sprache $L(FG, \lambda) = \{x \in L(FG) \mid f(x) = \lambda\}$.

Das beschriebene Vorgehen wird als *Fuzzy Reasoning* bezeichnet. Die dahinterstehenden Konzepte und Techniken sind bereits in verschiedenen Anwendungskontexten verwendet worden. Weitere Details über die verschiedenen Konzepte und Techniken sind in [Kas96] und [Nov92] beschrieben. Einen Überblick über Anwendungen der Techniken in verschiedenen Anwendungskontexten ist in [NAF03] und [FUZ03] beschrieben.

4.4.2 Wahl des initialen Genauigkeits- und Schwellwerts

Das Ergebnis der Ausführung eines Regelkatalogs - der annotierte abstrakte Syntaxgraph - ist stark abhängig von der Wahl der Genauigkeits- und der Schwellwerte. Der Genauigkeitswert einer Musterinstanzregel eines Regelkatalogs ist das Verhältnis der richtigen Ergebnisse zu allen gefundenen Ergebnissen, siehe Abbildung 4.13.

Der Genauigkeitswert ist also von den Ergebnissen des zu analysierenden Quelltextes abhängig und kann somit apriori nur geschätzt werden. Die Genauigkeit der Schätzung hängt dabei maßgeblich von der Erfahrung des Analysten und dem Vorwissen über den zu analysierenden Quelltext ab.

$$belief = 1 - \frac{|false\text{–}positives|}{|Ergebnisse|}$$

Abbildung 4.13 Wahl des initialen Genauigkeitswerts

Der Schwellwert einer Musterinstanzregel sollte so gewählt sein, dass er zu unpräzise Ergebnisse für die Analyse nicht weiter in Betracht zieht. Da die Präzision der Ergebnisse sich aus den Genauigkeitswerten der Musterinstanzregeln ergeben, entspricht die Angabe eines Schwellwerts quasi der Schätzung über eine Schätzung. Dies macht die Angabe eines initialen Schwellwerts für eine Musterinstanzregel sehr schwierig. Allerdings sollte der Schwellwert einer Musterinstanzregel maximal so groß sein wie die Genauigkeitswerte der abhängigen Musterinstanzregeln, da sonst das Story-Pattern der Musterinstanzregel nicht ausgeführt werden kann.

4.5 Regelerweiterungen

In der praktischen Anwendung der Graphersetzungssysteme hat sich herausgestellt, dass sich durch die Einführung der Erweiterungen in Graphtransformationsregeln die Anzahl der Musterinstanzregeln reduzieren lässt. Im Folgenden werden drei Erweiterungen für Musterinstanzregeln vorgestellt, die die Anzahl der Musterinstanzregeln eines Regelkatalogs reduzieren und sich in der praktischen Anwendung als nützlich erwiesen haben. Die vorgestellten Erweiterungen sind negative Elemente, Mengenknoten und OCL Ausdrücke mit Bedingungen, die die Konformität eines Regelkatalogs erweitern.

Die Erweiterungen haben direkte Entsprechungen in Story-Pattern, so dass die Abbildung der Musterinstanzregeln auf Story-Pattern hier nicht weiter ausgeführt wird. Die Konformitätsbedingungen aus Definition 4.10 gelten dabei auch für erweiterte Musterinstanzregeln, wobei negative Elemente nicht für den Zusammenhang des RegelASGausschnitts berücksichtigt werden dürfen. Der RegelASGausschnitt muss also ohne die negativen Elemente zusammenhängend sein.

Definition 4.12: Erweiterte Musterinstanzregel

Eine erweiterte Musterinstanzregel *rule = (RL, AL, SI, SG, an, bf, th, Neg, Multi, OCL)* ist eine Musterinstanzregel mit zusätzlichen

- $Neg \subset \{N_{SG}, E_{SG}\}$ negative Elemente
  - $\forall e = (src, el, i, q, tgt) \in E_{SG}$ *mit* $e \in Neg$ *gilt:* $src \notin Neg \wedge tgt \notin Neg$
- $Multi \subset \{N_{SG}\}$ Mengenknoten
  - $\forall n \in Multi$ *gilt:* $n \notin Neg$ *und* $\exists e = (src, el, i, q, tgt) \in E_{SG}$ *mit* $src \notin Multi$ *und* $Assocs_{SG}(el) = (srcNI, srcCard, atype, abasetype, tgtNI, tgtCard) \wedge instanceOf(n, tgtNI) = true \wedge tgtCard = many$
- *OCL endliche Menge von OCL-Ausdrücken* OCL-Ausdrücken

Die vorgestellten Erweiterungen erlauben es, die Anwendungsstelle eines Story-Patterns näher zu spezifizieren, und enthalten keine Graphtransformationen. Abbildung 4.14 und Abbildung 4.15 enthalten eine Delegations- und eine Bridge-Regel, die den Beispielregelkatalog aus Abbildung 4.1 erweitern. Das Regeldefinitionsdiagramm ist nicht mit aufgeführt, es wird aber erweitert, so dass jeweils eine Klasse Bridge und Composite erzeugt wird.

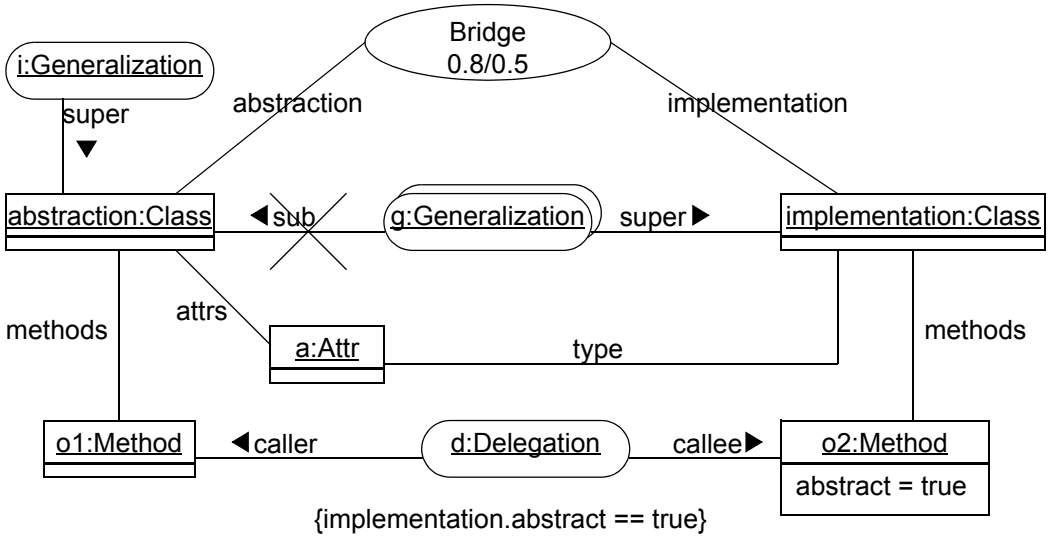


Abbildung 4.14 Bridge-Regel Beispiel

Negative Elemente einer Musterinstanzregel dürfen in der Anwendungsstelle des entsprechenden Story-Patterns nicht vorhanden sein. Dargestellt werden negative Elemente durch ein Kreuz des entsprechenden Elements des RegelASGausschnitts. In Abbildung 4.14 darf es also keine sub Kante geben.

Im Gegensatz zu negativen Elementen werden Mengenknoten dazu verwendet, alle möglichen Objekte mit in die Anwendungsstelle aufzunehmen. Mengenknoten werden durch

einen dreidimensionalen Rahmen des Knotens dargestellt. In der Bridge-Regel wird durch die Kombination des Mengenknotens g:Generalization und der negativen Kante sub erreicht, dass die Abstraktionsklassen, die an der Bridge beteiligt sind, keine Subklassen der Implementierungsklasse sind.

Durch die Angabe der OCL-Ausdrücke können zum Beispiel Attributbedingungen formuliert werden. Für eine detailliertere Beschreibung der zulässigen OCL-Ausdrücke sei auf [Zün01] verwiesen. Attributbedingungen werden in geschweiften Klammern notiert. Vergleiche mit festen Werten können auch in das Objekt direkt geschrieben werden, wodurch die Objektbezeichnung der Attributbedingung entfällt, siehe Abbildung 4.14.

Eine besondere Rolle spielen Navigationsausdrücke - auch *Pfade* genannt. Navigationsausdrücke sind OCL-Ausdrücke, die es erlauben, über eine Menge von Kanten zu navigieren, um von einem Knoten zu einem anderen zu gelangen. Dementsprechend sind sie Stellvertreter für Kanten im Graphen eines Story-Pattern.

Daraus ergibt sich eine Verfeinerung der Zusammenhangsbedingung für RegelASGausschnitte. Ein RegelASGausschnitt gilt als zusammenhängend, wenn im zugehörigen Story-Pattern alle Objekte des Story-Patterns ausgehend vom Musterinstanzobjekt über Links oder entsprechende Pfade erreichbar sind. Navigationsausdrücke werden durch einen gestrichelten Pfeil - ausgehend vom Startobjekt des Pfads zum Zielobjekt - dargestellt.

Die Delegations-Regel in Abbildung 4.15 verwendet einen Navigationsausdruck (astChildren)*, um im Methodenrumpf der caller Methode einen Methodenaufruf zu finden. Dabei sichert die Attributbedingung zu, dass der Name der Methode gleich dem Namen der callee Methode ist. Der Navigationsausdruck beschreibt dabei alle Wege, die über Kanten des abstrakten Syntaxbaums erreichbar sind.

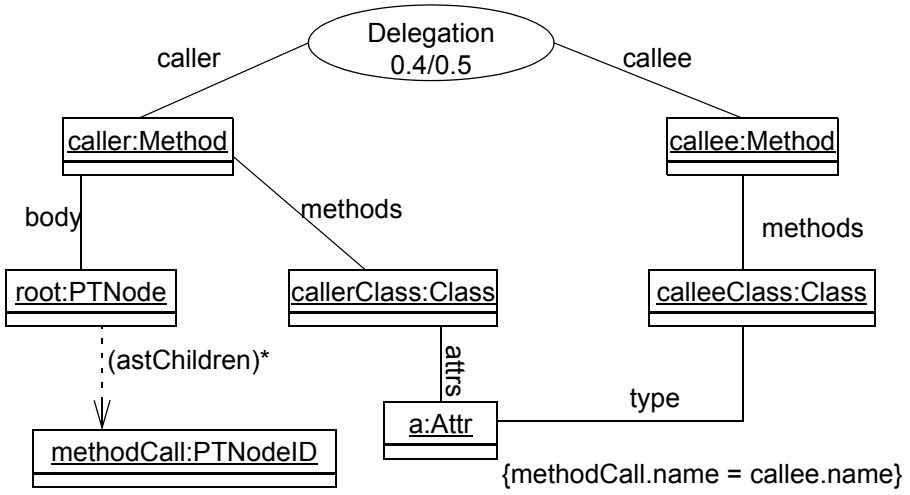


Abbildung 4.15 Delegation-Regel Beispiel

Der niedrige Genauigkeitswert der Delegation-Regel beruht auf der Tatsache, dass eine Delegation aufgrund von Namensäquivalenzen spezifiziert wird. Durch Techniken des Übersetzerbaus wäre eine genauere Spezifikation und damit ein höherer Genauigkeitswert möglich. Der Genauigkeitswert der Bridge-Regel ist ebenfalls niedrig, da die Musterinstanzregel nicht alle Anteile eines Bridge-Musters aus [GHJV95] enthält.

4.6 Zusammenfassung

Für die Erkennung von Entwurfsmusterinstanzen in Quelltexten ist ein Regelkatalog definiert worden, der aus einer Menge von Musterinstanzregeln und einem Regeldefinitionsdiagramm besteht. Die Definition erlaubt, dass eine Musterinstanzregel eine andere Musterinstanzregel verfeinert, und Musterinstanzregeln aus anderen Musterinstanzregeln komponiert werden können. Die Semantik einer Musterinstanzregel ist durch die Abbildung auf Story-Pattern definiert worden. Story-Pattern sind Graphtransformationsregeln, des in [Zün01] vorgestellten formalen Regelsystems.

Die verwendete Fuzzy Reasoning Technik basiert auf Fuzzymengen beziehungsweise Fuzzygrammatiken mit einer Minimum- und Maximumbildung zur Berechnung des Präzisionswerts eines Ergebnisses. Eine andere Fuzzy Reasoning Technik wird in [SK02] zur Rückgewinnung der Architekturinformationen verwendet. Zur Berechnung der Fuzzywerte wird ein nicht exaktes Patternmatching eingesetzt.

Die Story-Pattern, die für die Musterinstanzregeln eines Regelkatalogs erzeugt werden, sind mit den Parsingregeln von geschichteten Graphgrammatiken vergleichbar [RS97]. Geschichtete Graphgrammatiken sind dabei weniger restriktiv als die hier vorgestellten Musterinstanzregeln. In [RS95] wird ein Verfahren vorgestellt, das es ermöglicht, aus einer geschichteten Graphgrammatik automatisch einen Parser zu konstruieren. Der konstruierte Parser ist ähnlich den hier verwendeten Regelkatalogen, enthält allerdings keine Fuzzy Reasoning Techniken und der Parser arbeitet die erzeugten Parserregeln in den vorgegebenen Schichten der Grammatik ab. Er kann relevante Ergebnisse daher nicht frühzeitig produzieren.

Im Folgenden Kapitel wird ein Regelausführungsmechanismus vorgestellt, der Musterinstanzregeln auf einen abstrakten Syntaxgraphen eines Quelltexts anwendet und relevante Ergebnisse frühzeitig produziert.

KAPITEL 5: REGELAUSFÜHRUNGSMECHANISMUS

Die Semantik eines Regelkatalogs ist durch die Abbildung der einzelnen Musterinstanzregeln auf Graphtransformationsregeln definiert worden, siehe Kapitel 4. Die Ausführung der Graphtransformationsregeln bei Graphersetzungssystemen erfolgt meist bottom-up, was mit einem single-shot Ansatz vergleichbar ist. Aussagekräftige Ergebnisse werden erst nach einer vollständigen Analyse des Quelltextes produziert, wodurch ein solcher Ansatz nicht skaliert, siehe [Qui94].

In diesem Kapitel wird ein Regelausführungsmechanismus vorgestellt, der es ermöglicht, aussagekräftige Zwischenergebnisse zu produzieren, ohne den Quelltext vollständig zu analysieren. Die Zwischenergebnisse können somit frühzeitig vom Reengineer beurteilt werden, wodurch der Reengineer die weitere Analyse steuern kann.

Der folgende Abschnitt beschreibt die zugrunde liegende Idee des optimierten Regelausführungsmechanismus anhand eines Beispielkatalogs. Eine optimierte Regelausführung in Kombination mit Kontrollflüssen bildet die Grundlage für die FUJABA Entwicklungsumgebung, die das in [Zün01] vorgestellte formale Regelsystem implementiert. Auf Basis des formalen Regelwerks wird der Inferenzalgorithmus im folgenden Abschnitt beschrieben. Die Berechnung der Präzisionswerte der Ergebnisse erfolgt getrennt vom Inferenzalgorithmus und ist durch ein Fuzzy-Petrinetz beschrieben, das in einem separaten Abschnitt vorgestellt wird.

5.1 Regelausführungsreihenfolge

Ziel des Regelausführungsmechanismus ist es, aussagekräftige Zwischenergebnisse frühzeitig während der automatischen Analyse zu produzieren [NSW+02]. Der Reengineer hat dadurch die Möglichkeit, frühzeitig Ergebnisse zu beurteilen und kann gegebenenfalls in die automatische Analyse eingreifen.

In Bezug auf den hier vorgestellten Ansatz entsprechen aussagekräftige Ergebnisse Annotationen im Quelltext, die Entwurfsmusterinstanzen repräsentieren. Annotationen werden dabei durch entsprechende Musterinstanzregeln im Regelkatalog erzeugt.

Die Regelausführungsreihenfolge zur frühzeitigen Produktion relevanter Ergebnisse hängt von den Abhängigkeiten der Musterinstanzregeln untereinander ab, siehe Definition 4.9. Im Folgenden werden als *Vorgängerregeln* einer Musterinstanzregel *rule* diejenigen Musterinstanzregeln bezeichnet, von der die Musterinstanzregel *rule* abhängt.

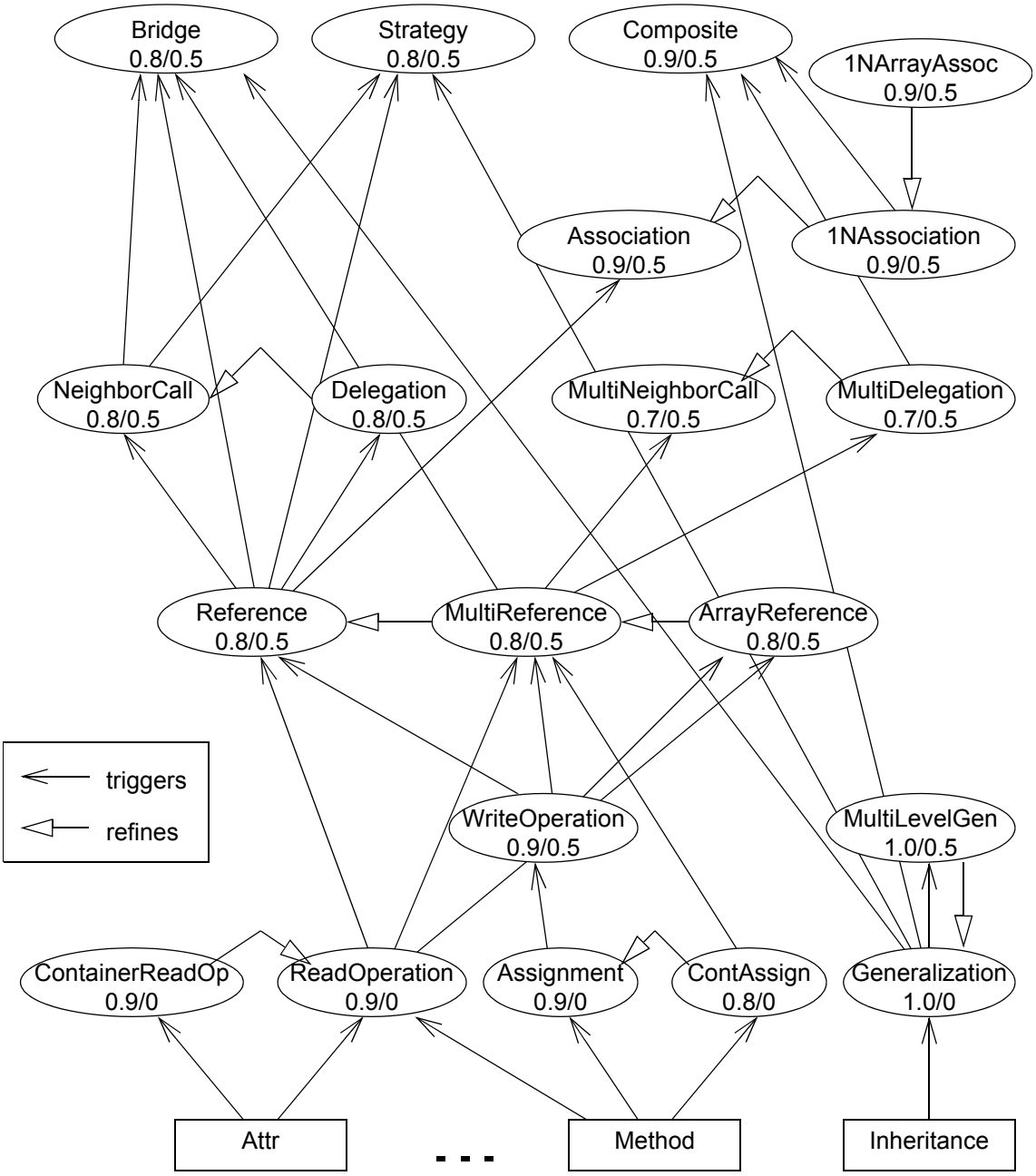


Abbildung 5.1 Regeltriggergraphbeispiel eines Regelkatalogs

Entsprechend werden als *Nachfolgerregeln* diejenigen Musterinstanzregeln bezeichnet, die von der Musterinstanzregel *rule* abhängig sind.

Zusätzlich zu den Abhängigkeiten einer Musterinstanzregel gemäß Definition 4.9 existieren für einen Regelkatalog Verfeinerungsbeziehungen, die bei der Regelausführungsrei-

henfolge berücksichtigt werden müssen. Sowohl die Abhängigkeiten als auch die Verfeinerungen können als Graph darstellt werden. Abbildung 5.1 zeigt einen solchen *Regeltriggergraph* (RTG), der aus einem Beispielkatalog erzeugt worden ist. Der zugehörige Regelkatalog ist zur Analyse der Abstract Windowing Toolkit (AWT) Bibliothek des Java Development Kits [JDK, AWT] eingesetzt worden, siehe [Wen01].

Aus Sicht der Regelausführungsreihenfolge werden Nachfolgerabhängigkeiten einer Regel als *Trigger* bezeichnet. Die Genauigkeitswerte und Schwellwerte einer Regel sind ebenfalls angegeben. Regeln werden aufgrund der natürlichen Abhängigkeitsreihenfolge schichtenweise angeordnet. Die Schichten beziehungsweise Regeln werden nummeriert. Regeln, die keine Vorgängerregeln haben, werden als *Rang-0* Regeln bezeichnet und erhalten die Nummer 0. Zusätzlich zu den Regeln beinhaltet ein RTG auch so genannte *Axiome*, die Klassen des abstrakten Syntaxgraphen entsprechen. Als Axiome werden alle Syntaxgraphklassen verwendet, deren Objekte in Rang-0 Regeln vorkommen. Triggerbeziehungen von Axiomen werden nur zu den entsprechenden Rang-0 Regeln gezogen.

Der Beispielregelkatalog, aus dem der RTG in Abbildung 5.1 konstruiert worden ist, dient zur Erkennung der Bridge-, Strategy- und Composite-Gamma-Muster. Die drei entsprechenden Regeln besitzen einen hohen Rang. Die Verfeinerungsbeziehungen zwischen Regeln aus dem Regeldefinitionsdiagramm sind ebenfalls durch entsprechende Kanten dargestellt.

Allgemeine Regelausführungssysteme arbeiten bottom-up aufgrund der Regelabhängigkeiten. Der RTG wird also prinzipiell in einer Art Breitensuche abgearbeitet. Bezogen auf das Beispiel werden dadurch erst die Regeln ContainerReadOp, ReadOperation, Assignment, ContAssign und Generalization solange es möglich ist angewendet, da diese Regeln direkt auf dem abstrakten Syntaxgraphen aufbauen und keine Vorgängerregeln haben. Anschließend werden die Regeln WriteOperation und MultiLevelGen angewendet und so weiter. Aussagekräftige Ergebnisse - Aussagen über Entwurfsmusterinstanzen - werden also erst spät produziert, weil die entsprechenden Regeln von annähernd allen anderen Regeln direkt oder transitiv abhängen, die vorher angewendet werden müssten.

Aussagekräftige Ergebnisse können früher produziert werden, wenn anstatt der Breitensuche eine Art Tiefensuche verwendet wird. Regeln, die Entwurfsmusterinstanzen repräsentieren, werden dadurch früher während der automatischen Analyse angewendet. Das Problem, bei einem solchen Vorgehen ist, gezielt Regeln auszuwählen, die auch anwendbar sind. Sind Regeln nicht anwendbar, weil Vorgängerregeln noch nicht angewendet worden sind, so erzeugt dies lediglich zusätzlichen Aufwand. Um diesen zusätzlichen Aufwand zu reduzieren, wird versucht, Vorgängerregeln durch ein top-down Vorgehen anzuwenden.

Ein reines top-down Vorgehen, bei dem alle möglichen Ausprägungen erzeugt werden und mit dem Quelltext validiert werden, ist praktisch nicht anwendbar, da der Suchraum selbst bei kleinen Regelkatalogen explodiert. Der Suchraum kann allerdings aufgrund der Regeln und der schon vorhandenen Annotationen reduziert werden, indem die vorhande-

nen Annotationen Kontextinformationen an die Vorgängerregeln weiterleiten. Aufbauend auf diesen Kontextinformationen können die Vorgängerregeln angewendet werden oder wiederum ein Kontext für weitere Vorgängerregeln bestimmt werden.

Die Regelausführungsreihenfolge ist eine Kombination aus bottom-up und top-down Vorgehen. Abbildung 5.2 zeigt eine Beispielausführungsreihenfolge. Der abstrakte Syntaxgraph ist unten durch das graue Rechteck mit einigen Objekten und Beziehungen dargestellt. Annotationen, die bereits aufgrund einer erfolgreichen Regelanwendung erzeugt worden sind, sind als schwarzes Oval mit entsprechenden Annotationen vorhanden. Gestrichelte Ovale repräsentieren Annotationen, die sich zur Zeit top-down im Vorgehen befinden. Die Pfeile entsprechen dem bottom-up beziehungsweise dem top-down Vorgehen. An den Pfeilen sind die entsprechenden Kontexte für die Regelanwendung dargestellt.

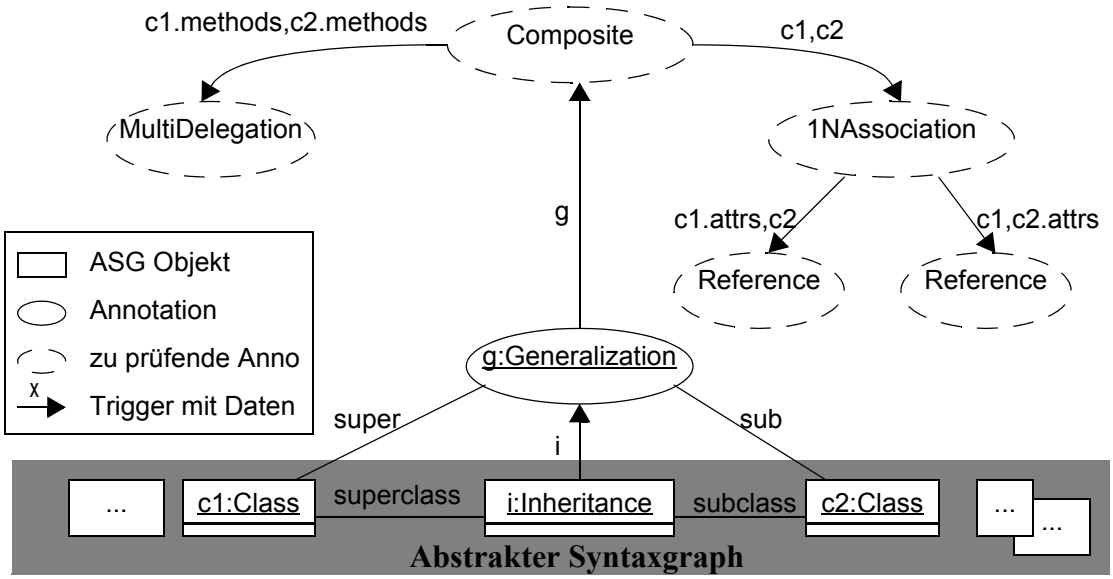


Abbildung 5.2 Beispielregelausführung

Der abstrakte Syntaxgraph (ASG) für die Analyse wird durch einen Parser erzeugt. Anschließend startet die automatische Analyse im bottom-up Modus, wobei initial alle Rang-0 Regeln durch entsprechende Objekte des abstrakten Syntaxgraphen getriggert werden. Die anfängliche Triggerung der Rang-0 Regeln ist ausreichend, um alle potenziellen Regelanwendungen, die während der Analyse erfolgen müssen, anzustoßen. Zusätzlich werden dadurch viele Fehler in top-down Vorgehen für Regeln mit hohem Rang vermieden, weil die notwendigen Kontexte nicht ausreichend sind. Beispielsweise ist der Suchraum in einem top-down Vorgehen für eine Composite-Regel, die nur von einer einzigen Klasse getriggert wird, zu groß.

Für ein Objekt des abstrakten Syntaxgraphen *o*, das eine Regel *R* triggert, wird ein Regel/Kontext Paar *R(o)* erzeugt und im bottom-up Modus in eine Prioritätsschlange eingefügt. Die Prioritätsschlange ist absteigend aufgrund des Rangs der Regel des Regel/Kontext Paares sortiert. Die Verwendung des Rangs einer Regel für die Sortierung in der Prioritätsschlange ist dabei lediglich eine Möglichkeit. Eine andere Sortierung ist ebenfalls möglich, wodurch der Algorithmus weiter optimiert werden kann.

Im bottom-up Modus wird das erste Regel/Kontext Paar aus der Prioritätsschlange entfernt. Im Beispiel von Abbildung 5.2 ist das *Generalization(i)*. Die Generalization-Regel ist direkt auf das abstrakte Syntaxgraphobjekt *i* anwendbar und erzeugt eine Generalization-Annotation *g*. Die Generalization-Annotation *g* annotiert die beiden zugehörigen Klassen *c1* und *c2* mit *super* und *sub*. Im Gegensatz zu Objekten des abstrakten Syntaxgraphen, die lediglich Rang-0 Regeln triggern, triggern neu erzeugte Annotationen Nachfolgerregeln. Im obigen Beispiel triggert die neue Generalization-Annotation *g* die MultiLevelGen-Regel sowie die drei Regeln Bridge, Composite und Strategy, die Entwurfsmustern entsprechen. Es werden also die Regel/Kontext Paare *MultiLevelGen(g)* *Bridge(g)*, *Composite(g)* und *Strategy (g)* in die Prioritätsschlange eingefügt.

Die Regel/Kontext Paare *Bridge(g)*, *Composite(g)* und *Strategy(g)* haben die gleiche hohe Rangnummer im RTG aus Abbildung 5.1. Bei der Beispielausführung in Abbildung 5.2 wird als nächstes das Regel/Kontext Paar *Composite(g)* ausgewählt. Zu diesem Zeitpunkt kann die Composite-Regel allerdings nicht angewendet werden, weil seine Vorgängerregeln (MultiDelegation und 1NAssociation) noch nicht ausgeführt worden sind.

Wenn eine Regel in einem Kontext angewendet werden soll, dessen Vorgängerregeln noch nicht im selben Kontext angewendet worden sind, wird in den top-down Modus gewechselt. Der top-down Modus arbeitet anstatt mit einer Prioritätsschlange mit einem Stack, der initial leer ist. Im top-down Modus wird versucht, aufgrund des gegebenen Kontextes die Vorgängerregeln anzuwenden. Durch den gegebenen Kontext ist der entstehende Suchraum sehr begrenzt, im Beispiel durch die neu erzeugte Annotation *g*.

Das Regel/Kontext Paar *Composite(g)*, dass im top-down Modus ausgeführt werden soll, erfordert, dass alle Vorgängerregeln (MultiDelegation und 1NAssociation) mit ihrem entsprechenden Kontext angewendet worden sind. Dass heißt, dass die entsprechenden Regel/Kontext Paare auf den Stack gelegt werden. Wenn diese Regeln ebenfalls Vorgängerregeln haben - wie im Beispiel die Reference-Regel - werden diese ebenfalls mit ihrem Kontext auf den Stack gelegt.

Im Beispiel von Abbildung 5.2 sind der Kontext für die MultiDelegation- und die 1NAssociation-Regel verschieden. Der Kontext für die 1NAssociation-Regel sind die beiden Klassen *c1* und *c2* und der Kontext für die MultiDelegation-Regel besteht aus zwei Methoden *m1* und *m2* der beiden Klassen *c1* und *c2*, siehe Abbildung 5.3. Da Klassen normalerweise mehrere Methoden besitzen, müssen alle Methoden als Kontext für die Regel untersucht werden. Die Suche kann allerdings bei der ersten erfolgreichen Anwendung der Regel in einem Kontext abgebrochen werden.

Kontexte werden aufgrund der Struktur einer Regel berechnet - hier die Composite-Regel. Da der Kontext für die MultiDelegation generell größer ist, wird mit der 1NAssociation fortgefahren. Die 1NAssociation hat wiederum die Reference-Regel als Vorgängerregel, so dass für die beiden benötigten Reference-Annotationen der 1NAssociation-Regel jeweils ein eigenes Regel/Kontext Paar auf den Stack gelegt wird.

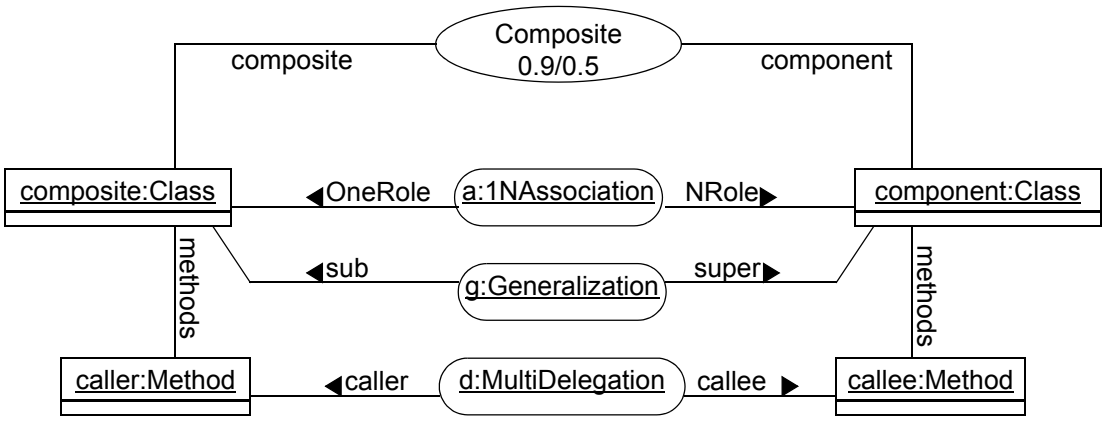


Abbildung 5.3 Composite-Regel Beispiel

Ein Regel/Kontext Paar an oberster Stelle des Stacks, das wiederum weitere Vorgängerregeln besitzt, wird dabei nicht vom Stack genommen, sondern es werden erst alle Vorgängerregeln mit ihrem entsprechenden Kontext auf den Stack gelegt. Dies Vorgehen erlaubt es, zielgerichtet Rang-0 Regeln zu erreichen.

Ist eine Regel im top-down Modus anwendbar, so erzeugt sie eine neue Annotation wie im bottom-up Modus. Sie wird vom Stack entfernt, und die nächste Regel wird versucht anzuwenden. Gleichzeitig wird die neu erzeugte Annotation als Kontext mit ihren triggernden Regeln in die bottom-up Prioritätsschlange eingefügt, weil sie Analyseergebnissen entsprechen, die im bottom-up Modus zu einem späteren Zeitpunkt sowieso erzeugt worden wären.

Der top-down Modus wird verlassen, wenn der Stack leer ist oder es keinen alternativen Kontext für eine Regel mehr gibt und alle Anwengungen fehlgeschlagen sind. Zum Beispiel sind alle Methoden einer Klasse alternative Kontexte für die MultiDelegation-Regel in Abbildung 5.2. Wenn für keine der Methoden eine MultiDelegation-Regel angewendet worden ist, wird der top-down Modus abgebrochen.

Ist der Stack leer, so ist die Regel, die den top-down Modus aktiviert hat, erfolgreich angewendet worden und eine entsprechende Annotation ist erzeugt worden; im Beispiel eine Composite-Annotation. Ist kein alternativer Kontext mehr für eine Regel vorhanden, so kann der top-down Modus beendet werden, und die Regel, die den top-down Modus aus-

gelöst hat, konnte nicht angewendet werden. In beiden Fällen wird wieder im bottom-up Modus fortgefahren.

Aufgrund der beschriebenen Regelausführungsreihenfolge entspricht jede erzeugte Annotation einem Teilergebnis, das im weiteren Verlauf der Analyse nicht wieder zurückgenommen wird. Lediglich bei der sich anschließenden Berechnung des Präzisionswerts einer Annotation kann es sein, dass ein Teilergebnis nicht weiter berücksichtigt wird. Prinzipiell kann also die Ausführung jederzeit angehalten werden. In der Praxis hat sich allerdings herausgestellt, dass eine Unterbrechung im top-down Modus nicht sinnvoll ist, da der Reengineer keine Informationen über den aktuellen Zustand des Stacks und den Ablauf des top-down Modus hat.

Wenn die Prioritätsschlange des bottom-up Modus leer ist, sind alle Annotationen aufgrund von erfolgreichen Regelausführungen erzeugt worden. Die Analyse ist damit beendet.

5.2 Optimierte Regelanwendung

Die Ausführung einer Graphtransformationsregel enthält das Graph-Isomorphie-Problem, das NP-vollständig ist. Durch die Angabe eines Kontextes zu einer Regel kann im durchschnittlichen Fall allerdings die Laufzeit einer Regelanwendung auf polynomielle und in vielen Fällen auf lineare Zeit reduziert werden. Diese Optimierung der Regelanwendung durch Angabe eines Kontextes wird im ersten Teil dieses Abschnittes beschrieben. Der zweite Teil stellt in Auszügen das formale Regelsystem aus [Zün01] vor, dass für die Spezifikation des Inferenzalgorithmus in den folgenden Abschnitten verwendet wird.

Die Anwendung einer Graphtransformationsregel auf einen Graphen - den Wirtsgraphen - besteht im Wesentlichen aus zwei Teilen. Im ersten Teil wird eine Anwendungsstelle für die Regel gesucht, und im zweiten Teil werden die Modifikationen durchgeführt. Das Suchen einer Anwendungsstelle entspricht dem NP-vollständigen Problem der Teilgraphensuche, vgl. [Meh84]. Demgegenüber ist der Aufwand für durchzuführende Modifikationen zu vernachlässigen.

5.2.1 Einschränkung der Anwendungsstellen

Eine Möglichkeit, den Aufwand für die Teilgraphensuche bei der Anwendung einer Graphtransformationsregel zu verringern ist, die Anzahl der möglichen Anwendungsstellen einzuschränken. Die Einschränkung der Anwendungsstellen wird durch die Angabe eines Teilgraphen des Wirtsgraphen, auf den die Regel angewendet werden soll, erreicht.

Die Angabe des Teilgraphen, auf dem eine Regel angewendet werden soll, kann auf verschiedene Arten erreicht werden, vergleiche [Zün95, SWZ95, FNTZ98]. So können Teilgraphen explizit durch eine Menge von Knoten und Kanten angegeben oder durch Berechnungsvorschriften und Filter auf dem Wirtsgraphen festgelegt werden.

Eine weitere Möglichkeit, die in der Praxis eingesetzt wird, ist den Teilgraphen durch eine Menge von Knoten zu beschreiben, wobei die Knoten an Knoten der Regel gebunden werden. Knoten einer Regel, an die Knoten im Graphen gebunden werden sollen, werden explizit angegeben und als *gebundene Knoten* oder im Kontext von Story-Pattern[1] als *gebundene Objekte* bezeichnet.

Das Story-Pattern in Abbildung 5.4 ist ein Teilausschnitt des aus der Composite-Regel abgeleiteten Story-Pattern aus Abbildung 4.10. Die gebundenen Objekte sind c1 und c2. Bei gebundenen Objekten wird der Name fett dargestellt, und der Typ der Objekte entfällt in der Darstellung, obwohl sie aufgrund des objektorientierten Graphen Instanz einer Klasse sind. Im Beispiel repräsentieren die Objekte c1 und c2 jeweils eine Klasse im abstrakten Syntaxgraphen.

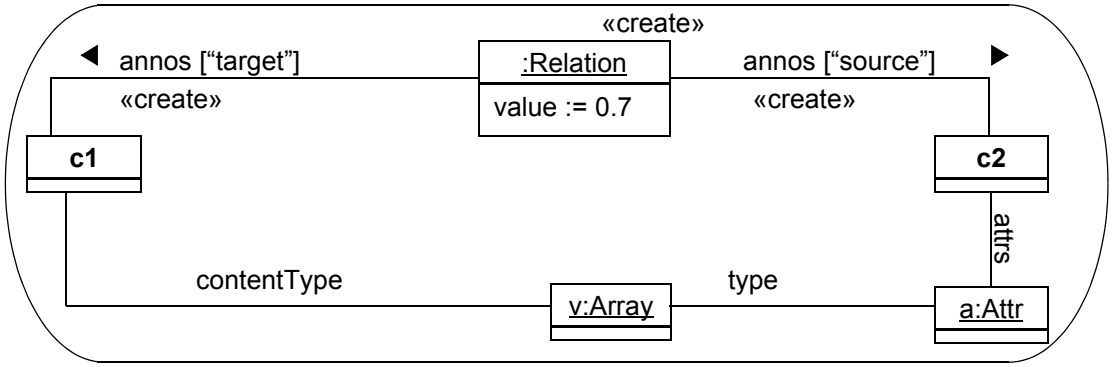


Abbildung 5.4 Story-Pattern Beispiel mit gebundenen Objekten

Durch die Angabe der beiden Objekte c1 und c2 sind bei der Regelausführung lediglich alle Attribute der Klasse, repräsentiert durch das Objekt a:Attr, zu überprüfen. Dabei wird überprüft, ob ein Attribut ein Array ist und der Inhalt des Arrays Klassen sind, die durch das Objekt c1 repräsentiert sind. Wären c1 und c2 keine gebundenen, sondern so genannte *freie Objekte*, so müssten bei der Ausführung der Regeln alle Klassen im abstrakten Syntaxgraphen überprüft werden.

5.2.2 Angabe des Anwendungskontextes

Die Einschränkung der Anwendungsstelle einer Regel erfolgt durch die Angabe gebundener Objekte in der Regel. Offen ist, an welche Knoten des Wirtsgraphen - der Anwendungskontext - die Objekte gebunden werden, und wie der Anwendungskontext angegeben werden kann.

Eine Möglichkeit, den Anwendungskontext einer Regel anzugeben ist, einzelne Knoten im Graphen zu markieren. Dies kann zum Beispiel durch spezielle Markierungsknoten er-

1. Graphtransformationsregeln werden im formalen Regelwerk in [Zün01] als Story-Pattern bezeichnet.

reicht werden. Sind die Markierungsknoten eindeutig, so schränkt dies die Anwendungsstellen der Regel ein. Werden die Markierungsknoten in der Graphtransformationsregel an andere Knoten gehängt, so können Kontrollflüsse und Datenflüsse festgelegt werden. Dadurch ist es möglich, mit Graphtransformationsregeln zu programmieren. Die zugehörigen Systeme werden als *Programmierte Graphersetzungssysteme* bezeichnet, siehe [Zün95].

Programmierte Graphersetzungssysteme wie PROGRES [SWZ95, Zün95] oder FUJABA [FNTZ98, KNNZ99, NNZ00, KNNZ00] ermöglichen es, Kontrollflüsse explizit durch entsprechende Sprachkonstrukte zu definieren. Datenflüsse können durch Modifikationen von Graphelementen und parametrisierte Regeln explizit festgelegt werden beziehungsweise sind aufgrund von Namesräumen implizit vorhanden.

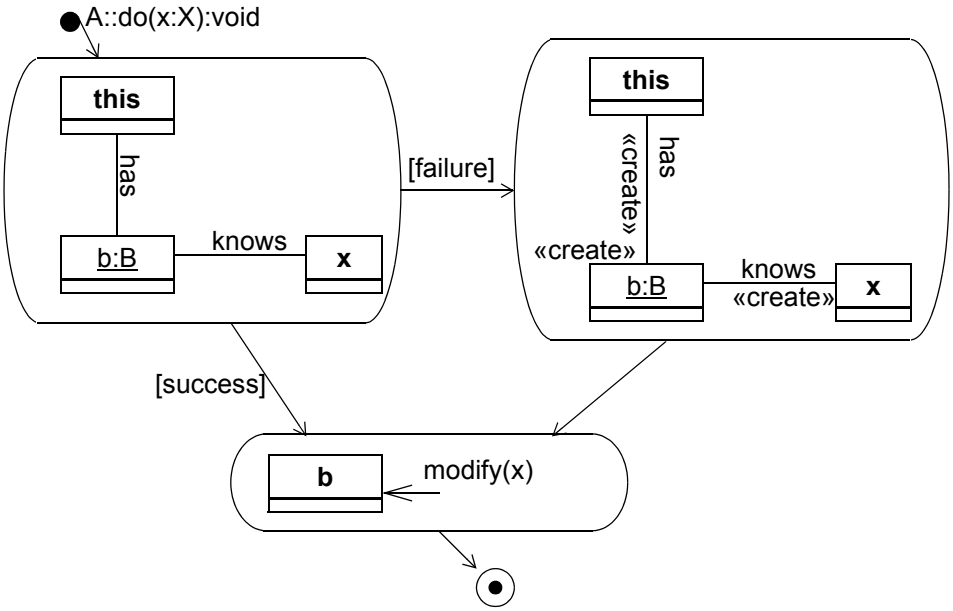


Abbildung 5.5 Story-Diagramm Beispiel

Das in [Zün01] vorgestellte Graphersetzungssystem ist die formale Grundlage der FUJABA Entwicklungsumgebung. Kontrollflüsse werden durch UML-Aktivitätendiagramme beschrieben. Story-Pattern sind einzelne Aktivitäten. Die Kombination wird als *Story-Diagramm* bezeichnet und ist gleichzeitig ein Namensraum. Ein Story-Diagramm ist eine Methode einer Klasse. Methoden haben einen Namen, eine Menge von Parametern sowie einen Rückgabeparameter. Durch Kollaborationsanweisungen können Methoden auf anderen Objekten aufgerufen werden.

Das Beispiel in Abbildung 5.5 zeigt das Story-Diagramm der Methode do der Klasse A. Der Methode wird ein Objekt x als Parameter übergeben und hat keinen Rückgabeparameter (void). Der Kontrollfluss beginnt bei der Startaktivität - dargestellt als ausgefüllter Kreis - und endet bei einer Stopaktivität - dargestellt als doppelter Kreis. Das Story-Pat-

tern oben links wird als erstes ausgeführt. Ist eine Anwendungsstelle des Story-Pattern gefunden und sind alle Modifikationen ausgeführt worden - die Ausführung des Story-Pattern war erfolgreich - so wird die success Kante verfolgt und das untere Story-Pattern wird ausgeführt. Im Fall eines Fehlschlags des ersten Story-Pattern wird das Story-Pattern oben rechts ausgeführt.

Die beiden oberen Story-Pattern enthalten jeweils zwei gebundene Objekte this und x. Der Anwendungskontext für x, also ein existierender Knoten im Wirtsgraphen, wird der Methode als Parameter übergeben. Der Anwendungskontext von this ist das aktuelle Objekt, auf dem die Methode aufgerufen wird. Der Anwendungskontext für das gebundene Objekt this ist also implizit, während der Anwendungskontext für das Objekt x explizit ist. Dadurch, dass ein Story-Diagramm einen Namensraum beschreibt, kann im unteren Story-Pattern das Objekt b als Anwendungskontext verwendet werden. Der Anwendungskontext ist durch eines der oberen Story-Pattern festgelegt. Zusätzlich enthält das untere Story-Pattern einen Methodenaufruf modify auf dem Objekt b, bei dem das Objekt x als Parameter übergeben wird.

5.2.3 Ausführung einer Regel

Zusätzlich zur Angabe eines Anwendungskontextes einer Regel kann die Teilgraphensuche bei der Ausführung einer einzelnen Regel algorithmisch optimiert werden. Durch die algorithmische Optimierung kann der Aufwand für die Teilgraphensuche im durchschnittlichen Fall auf polynomielle beziehungsweise lineare Laufzeit verringert werden.

Bei der Teilgraphensuche wird ausgehend von den gebundenen Objekten versucht, über die Verbindungen im Story-Pattern entsprechende Objekte im Wirtsgraphen zu erreichen. Sollte ein Objekt nicht vorhanden sein, so kann die Teilgraphensuche an einer anderen Stelle fortgesetzt werden. Dabei arbeitet die Teilgraphensuche nach dem Force-Failure-Prinzip. Das Force-Failure-Prinzip ist ursprünglich eine Optimierung bei Backtracking-Algorithmen gewesen, bei der versucht wird, gezielt möglichst große Teilbäume nicht abarbeiten zu müssen. Weitere optimierte Backtracking-Verfahren werden in [Kum92] vorgestellt.

Angewandt auf das Beispiel in Abbildung 5.6 wird beim Force-Failure-Prinzip, ausgehend vom aktuellen Objekt this, zuerst versucht, einen Knoten des Wirtsgraphen an das Objekt b:B zu binden. Ein solcher Knoten im Wirtsgraphen muss vom this Objekt über einen has-Link erreichbar sein. Im nächsten Schritt wird dann versucht, auch einen Knoten an das Objekt x:X zu binden. Existiert im Wirtsgraph kein Knoten, der an das Objekt b:B gebunden werden kann, so kann die Teilgraphensuche abgebrochen werden, weil durch die has-Assoziation maximal ein Objekt erreicht werden kann. Würde anstatt dessen zuerst versucht, einen Knoten des Wirtsgraphen an das Objekt x:X zu binden, obwohl es keinen Knoten im Wirtsgraphen gibt, der an das Objekt b:B gebunden werden kann, so müsste dies für alle möglichen Bindungen an das Objekt x:X überprüft werden. Die Teilgraphensuche würde dadurch ineffizienter.

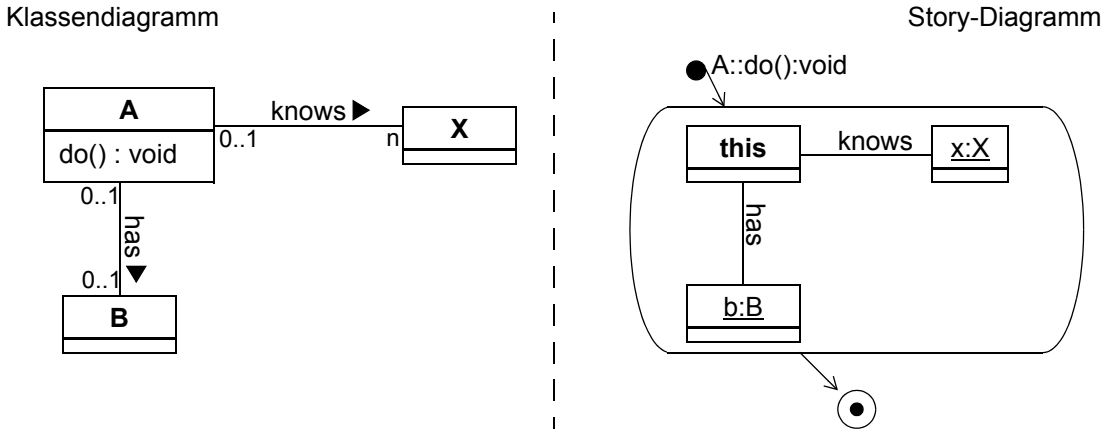


Abbildung 5.6 Story-Diagramm Beispiel für optimierte Teilgraphensuche

Eine optimale Ausnutzung des Force-Failure-Prinzips kann dabei erst vor der Regelausführung ermittelt werden, weil der aktuelle Wirtsgraph und insbesondere die Anzahl zu erreichender Knoten eine wichtige Rolle spielt. Dies bedeutet eine Analyse des Wirtsgraphen vor jeder Regelausführung, was den Gesamtaufwand für die Teilgraphensuche erhöht. In der Praxis wird aufgrund der Kardinalitäten der Assoziationen eine Reihenfolge der Bindungen für jede Regel festgelegt, siehe [FNTZ98].

5.3 Inferenzalgorithmus

Der Inferenzalgorithmus, der die Musterinstanzregeln eines Regelkatalogs gemäß der in 5.1 beschriebenen Ausführungsreihenfolge unter Verwendung der in 5.2 beschriebenen optimierten Regelanwendung ausführt, wird in diesem Abschnitt beschrieben. Zur Definition des Inferenzalgorithmus wird das formale Regelsystem aus [Zün01] verwendet, das der Entwicklungsumgebung FUJABA [FNTZ98] zugrunde liegt. Verwendete Sprachelemente werden informal beschrieben. Für Details und eine formale Beschreibung sei auf [Zün01] verwiesen.

5.3.1 Konstruktion Regeltriggergraph

Die Reihenfolge der Regelausführung im Inferenzalgorithmus wird durch den Regeltriggergraphen bestimmt, der aus einem Regelkatalog konstruiert wird. Der Regeltriggergraph ist ein wohlgeformter, objektorientierter Graph, dessen Schemainformation in Abbildung 5.7 dargestellt ist.

Der Graph besteht allgemein aus Axiomen und Regeln. Axiome entsprechen Klassen des abstrakten Syntaxgraphen, deren Objekte in Regeln verwendet werden, die keine Vorgängerregel haben (Rang-0 Regeln). Für jede Musterinstanzregel eines Regelkatalogs wird ein Objekt der Klasse Rule erzeugt und der Genauigkeits- und der Schwellwert der Musterinstanzregel in das entsprechende Objekt übernommen. Sowohl Axiom Objekte wie

auch Rule Objekte werden über einen Namen identifiziert. Der Name ist dabei eindeutig, weil der abstrakte Syntaxgraph ein wohlgeformter, objektorientierter Graph ist, und die Namen der Musterinstanzregeln eines Regelkatalogs eindeutig sind. Zusätzlich müssen die Namen der Musterinstanzregeln disjunkt zu den Namen der Klassen des abstrakten Syntaxgraphen sein. Gegebenenfalls müssen diese umbenannt werden.

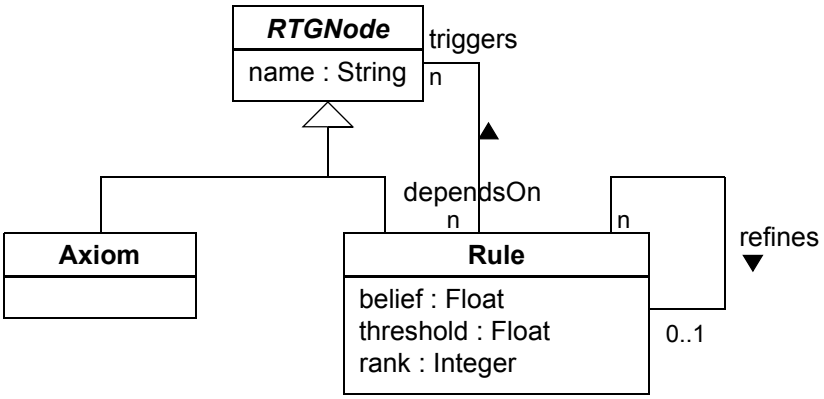


Abbildung 5.7 Schemainformation für Regeltriggergraphen

Die refines-Assoziation der Klasse Rule ist eine direkte Entsprechung der *rf* Funktion eines Regelkatalogs, siehe Definition 4.7. Die Struktur des Teilgraphen, der aus den Rule Objekten und den refines-Links besteht, ist dabei isomorph zu dem Teil der erweiterten Schemainformation aus Definition 4.8, der die Annotation-Klassen durch die Vererbungsstruktur beinhaltet.

Die triggers/dependsOn-Assoziation entspricht weitestgehend der Definition der Regelabhängigkeit gemäß Definition 4.9. Für zwei Musterinstanzregeln *r1* und *r2* eines Regelkatalogs, die voneinander abhängig sind, wird ein triggers/dependsOn-Link im Regeltriggergraphen erzeugt. Zusätzlich werden Axiom-Objekte mit ihren Rang-0 Regeln über triggers/dependsOn-Links verbunden. Dass heißt, ist ein Objekt des abstrakten Syntaxgraphen in einer Rang-0 Regel vorhanden, so wird ein triggers/dependsOn-Link vom entsprechenden Axiomobjekt zum Regelobjekt erzeugt. Mehrfache triggers/dependsOn-Links werden dabei verschmolzen. Im Folgenden werden diese Links entweder als triggers-Link oder als dependsOn-Link bezeichnet, um ihre Traversierungsrichtung festzulegen. Im Beispiel von Abbildung 5.1 triggert eine WriteOperation-Regel eine Reference-Regel, MultiReference-Regel und ArrayReference-Regel, während sie von der Assignment-Regel abhängt und dadurch von ihr getriggert wird.

5.3.2 Regelränge

Für die Regelausführungsreihenfolge im Inferenzalgorithmus ist der Rang einer Regel ausschlaggebend. Der Rang einer Regel muss eindeutig sein und kann zur Optimierung der Regelausführungsreihenfolge eingesetzt werden.

Definition 5.1: Rang einer Regel

Der Rang einer Regel ist durch die Funktion $rank : R \rightarrow \mathbb{R}$ für einen Regelkatalog *RK=(rSI, R, rf)* gegeben.

Regelränge unterliegen allerdings Bedingungen, die für den Inferenzalgorithmus notwendig sind. Dabei wird verlangt, dass der Rang einer Regel *r* höher ist als alle Ränge der Regeln, die Annotationen erzeugen, die für die Anwendung der Regel *r* verwendet werden können.

Definition 5.2: Bedingungen für Regelränge

Die Regelränge eines Regelkatalogs *RK = (rSI, R, rf)*, gegeben durch $rank : R \rightarrow \mathbb{R}$ müssen folgende Bedingungen erfüllen:

- $rank(ruleA) > rank(ruleB)\ \forall ruleA, ruleB \in R$ *mit dep (ruleA, ruleB) = true*
- $rank(ruleA) > rank(ruleB)\ \forall ruleA, ruleB \in R$ *mit dep(ruleA,rn) = true*
  und rf(ruleB)=r2 und rf(r2)=r3 und ... und rf(rn-1)=rn, $\forall r2,...,rn \in R$

Für den Inferenzalgorithmus wird im Folgenden der natürliche Rang einer Regel, bezogen auf ihre Abhängigkeit zu anderen Regeln, im Regelkatalog verwendet.

Definition 5.3: Natürlicher Rang einer Regel

Der natürliche Rang wird durch die Funktion $natRank : R \rightarrow \mathbb{R}$ für einen Regelkatalog *RK=(rSI, R, rf)* definiert mit:

- *natRank (rule) = 0 wenn* $\neg\exists\ r1 \in R$ *mit dep (rule, r1) = true*
- *natRank (rule) = 1 + max {rank(r) | dep (rule, r) = true} sonst*

Für alle Regeln, die keine Vorgängerregeln haben, ist der Rang definiert als 0 (Rang-0 Regeln). Alle anderen Regeln erhalten als Rang den maximalen Rang ihrer Vorgängerregeln plus eins. Der natürliche Rang für alle Regeln eines Regeltriggergraphen kann damit nach der Konstruktion durch einen einfachen rekursiven Algorithmus bestimmt werden.

Für den natürlichen Rang ist die erste Bedingung für Regelränge aus Definition 5.2 aufgrund seiner Definition immer erfüllt. Die zweite Bedingung ist Regelkatalog spezifisch und muss damit für jeden Regelkatalog beziehungsweise Regeltriggergraphen überprüft werden. Die Bedingung besagt, dass alle Nachfolgerregeln einer Regel einen höheren Rang haben müssen als alle Verfeinerungen der Regel. Im Beispielregeltriggergraph in Abbildung 5.1 ist diese Bedingung erfüllt. Die Bedingung wäre nicht mehr erfüllt, wenn die MultiLevelGen-Regel zusätzlich von der Delegation-Regel abhängig wäre, weil der natürliche Rang der MultiLevelGen-Regel gleich dem Rang der Bridge-, Strategy- oder Composite-Regel wäre.

5.3.3 Datenstrukturen

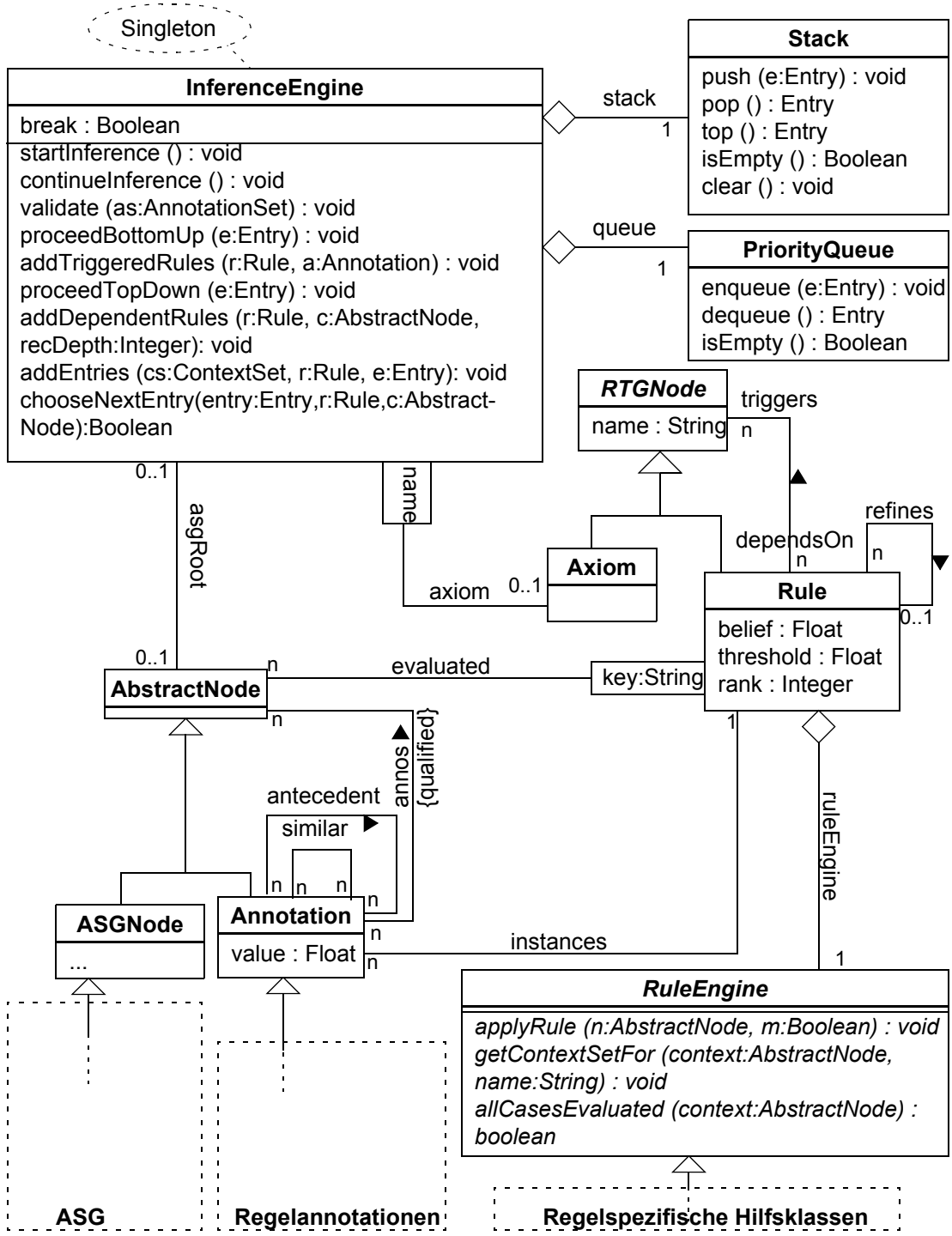


Abbildung 5.8 Klassendiagramm für den Inferenzalgorithmus

Die Datenstrukturen für den Inferenzalgorithmus bestehen aus der verfeinerten Schemainformation für einen wohlgeformten, objektorientierten, annotierbaren abstrakten Syntaxgraphen (Abbildung 4.9) und der Schemainformation des Regeltriggergraphen (Abbildung 5.7).

Das Klassendiagramm in Abbildung 5.8 beschreibt die Datenstrukturen des Inferenzalgorithmus basierend auf dem formalen Regelsystem aus [Zün01]. Die Schemainformation des Regeltriggergraphen ist auf der rechten Seite, und die Schemainformation des annotierbaren abstrakten Syntaxgraphen ist auf der linken Seite dargestellt. Klassen des verwendeten abstrakten Syntaxgraphen sind dabei nicht dargestellt. Ebenso sind die erzeugten Klassen für Musterinstanzregeln gemäß Abbildung 4.9 lediglich durch Rechtecke angedeutet, da sie vom jeweiligen Regelkatalog abhängen. Ebenso sind die regelspezifischen Hilfsklassen, die von der abstrakten Klasse RuleEngine erben, nicht mit aufgeführt.

Die Hilfsklassen werden im Inferenzalgorithmus für die optimierte Regelanwendung verwendet und besitzen entsprechende Methoden. Es existiert dabei für jede Musterinstanzregel eine Hilfsklasse, die durch die Aggregationsbeziehung ruleEngine mit dem Objekt des Regeltriggergraphen, das die Musterinstanzregel repräsentiert, verbunden ist.

Die qualifizierte evaluated-Assoziation zwischen der Klasse Rule des Regeltriggergraphen und der Klasse AbstractNode des abstrakten Syntaxgraphen speichert sowohl erfolgreiche wie auch erfolglose Regelanwendungen auf einen Kontext. Dadurch wird jedes Objekt des abstrakten Syntaxgraphen nicht mehrfach als Kontext für dieselbe Regel verwendet.

Wird eine Regel erfolgreich angewendet und eine Annotation erzeugt, wird die neu erzeugte Annotation an die Regel durch die Assoziation instances gebunden. Diese Assoziation wird für den schnellen Zugriff auf alle Annotationen, die von einer Regel erzeugt worden sind, verwendet.

Annotationen werden durch antecedent und similar-Assoziationen verbunden. Ein antecedent-Link korrespondiert mit dem dependsOn-Link der Regeln, und similar-Links korrespondieren mit refines-Links im Regeltriggergraphen. Die Beziehungen werden verwendet, um die Präzisionswerte im Anschluss an die Inferenz zu berechnen.

Der Inferenzalgorithmus wird durch die Klasse InferenceEngine repräsentiert. Das Attribut break wird verwendet, um den Algorithmus durch den Reengineer anhalten zu können. Die Methoden der Klasse beschreiben den Inferenzalgorithmus und werden in den folgenden Abschnitten im Detail beschrieben. Die Annotation Singleton an der Klasse repräsentiert das Gamma-Muster Singleton und besagt, dass nur ein Objekt der Klasse während der Laufzeit des Systems vorhanden ist. Dadurch wird sichergestellt, dass keine zweite Inferenz gestartet werden kann, die sich mit der ersten überschneidet. Die Assoziation asgRoot verbindet den abstrakten Syntaxgraphen mit dem Inferenzalgorithmus. Die qualifizierte axioms-Assoziation, die es erlaubt, über den Namen des Axioms auf die ent-

sprechende Klasse zuzugreifen, verbindet den Algorithmus mit dem Regeltriggergraphen und wird für einen effizienten Start des Inferenzalgorithmus verwendet.

Für den bottom-up Modus verwendet der Inferenzalgorithmus eine Prioritätsschlange, die durch die Klasse PriorityQueue repräsentiert wird und durch die queue-Assoziation erreichbar ist. Entsprechendes gilt für den Stack, der im top-down Modus verwendet wird. Sowohl die Prioritätsschlange wie auch der Stack sind Standarddatentypen mit entsprechenden Zugriffsmethoden.

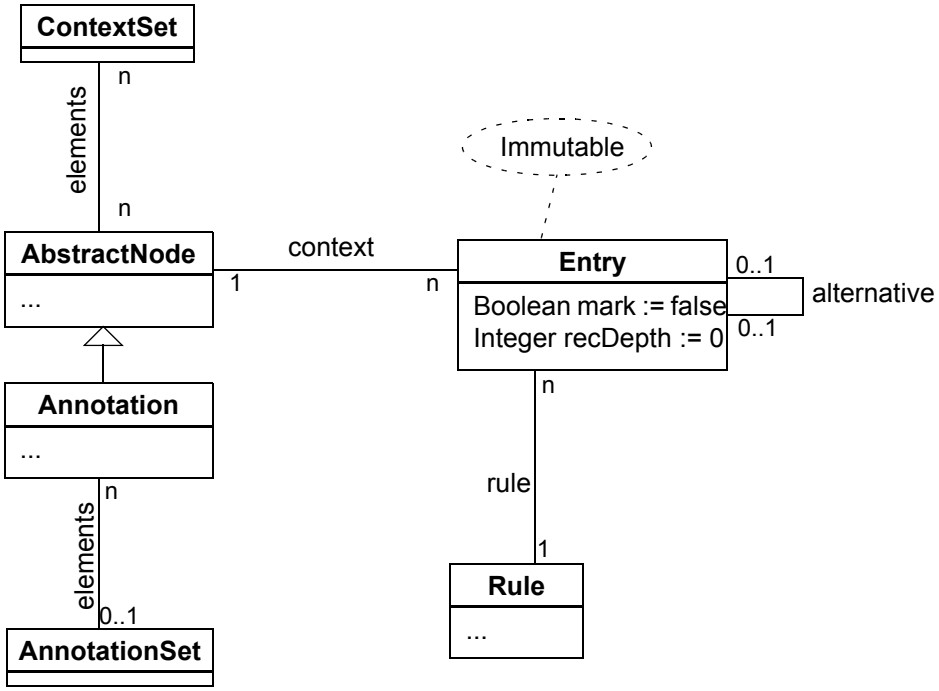


Abbildung 5.9 Zusätzliche Datenstrukturen für Inferenzalgorithmus

Sowohl die Prioritätsschlange wie auch der Stack verwalten als Elemente Objekte der Klasse Entry. Die Klasse ist mit anderen Hilfsdatenstrukturen für den Inferenzalgorithmus in Abbildung 5.9 dargestellt. Die Annotation Immutable besagt, dass die Werte eines Objekts nach seiner Erzeugung nicht nachträglich geändert werden dürfen. Diese Bedingung bezieht sich sowohl auf Attribute des Objekts wie auch auf Links des Objekts zu anderen Objekten. Nach einer Erzeugung eines Entry Objekts können also der Kontext context und die Regel rule nicht mehr geändert werden. Die alternative-Assoziation unter Entry Klassen wird im top-down Modus verwendet, um alternative Kontexte für Regeln zu verwalten. Die Prioritätsschlange kann durch die rule-Assoziation der Entry Klasse auf den Rang der Regel zugreifen und das Objekt entsprechend einsortieren. Für den Stack werden die Ränge nicht berücksichtigt allerdings wird in jedem Entry-Objekt die Rekursionstiefe in dem das Objekt erzeugt worden ist gespeichert. Dadurch kann der top-down Modus frühzeitig abgebrochen werden.

Die Hilfsdatenstrukturen AnnotationSet und ContextSet verwalten eine Menge von Annotationen beziehungsweise Objekten des abstrakten Syntaxgraphen. Alternativ könnten an dieser Stelle auch generische Datenstrukturen verwendet werden.

Im Folgenden werden die Methoden der Klasse InferenceEngine beschrieben. Methoden bestehen aus Kontrollfluss in Form von UML-Aktivitätsdiagrammen und Story-Pattern, die eine Aktivität beschreiben. Anschließend wird die Konstruktion der Methoden der regelspezifischen Hilfsklassen erläutert.

5.3.4 Schnittstelle für Reengineer

Aufgrund des Analyseprozesses in Kapitel 3 hat der Reengineer drei Möglichkeiten, den Inferenzalgorithmus anzustoßen. Er kann die Inferenz neu starten, fortsetzen oder selbst erzeugte Annotationen (Hypothesen) überprüfen lassen. Die entsprechenden Methoden sind startInference, continueInference und validate. Außerdem kann er den Algorithmus durch Setzen der break Variable unterbrechen.

Inferenz neu starten

Wird der Inferenzalgorithmus neu gestartet, siehe Abbildung 5.10, so werden im ersten Schritt der Methode (1) alle Annotationen gelöscht. Das Löschen besteht aus dem Sammeln aller Annotationen am abstrakten Syntaxgraphen durch einen Pfadausdruck und einer entsprechenden «destroy» Anweisung am Objekt a:Annotation. Durch den doppelten Rahmen werden dabei alle Annotationsobjekte erfasst. Die nächsten beiden Schritte dienen dazu, die Prioritätsschlange initial mit Elementen zu füllen.

Schritt (2) der Methode startInference sammelt alle Blätter des abstrakten Syntaxgraphen mittels eines Pfadausdrucks auf. Das * im Pfadausdruck repräsentiert den transitiven Abschluss über alle Kanten, die zur abstrakten Syntaxbaumrepräsentation des Quelltextes gehören. Gegebenenfalls muss dies bei der Verwendung eines anderen abstrakten Syntaxgraphen angepasst werden. Der doppelte Rahmen um das Story-Pattern bedeutet, dass es auf alle möglichen Objekte angewendet wird. Dabei wird für jedes gefundene Objekt das Story-Pattern (3) ausgeführt.

Story-Pattern (3) erzeugt ein e:Entry Objekt und fügt es in die Prioritätsschlange enqueue (e) ein. Dabei ist das im vorherigen Story-Pattern ermittelte Blatt des abstrakten Syntaxgraphen a der Kontext, und die zugehörige Regel rule wird über das entsprechende Axiom ax:Axiom und den triggers-Link ermittelt. Die ermittelte Regel muss eine Rang-0 Regel sein.

Nach dieser Initialisierungsphase wird der eigentliche Inferenzalgorithmus aufgerufen, Schritt (4). Das Starten der Inferenz und die Fortsetzung ist also, bis auf die Initialisierung der Prioritätsschlange und das Löschen aller Annotationen, identisch.

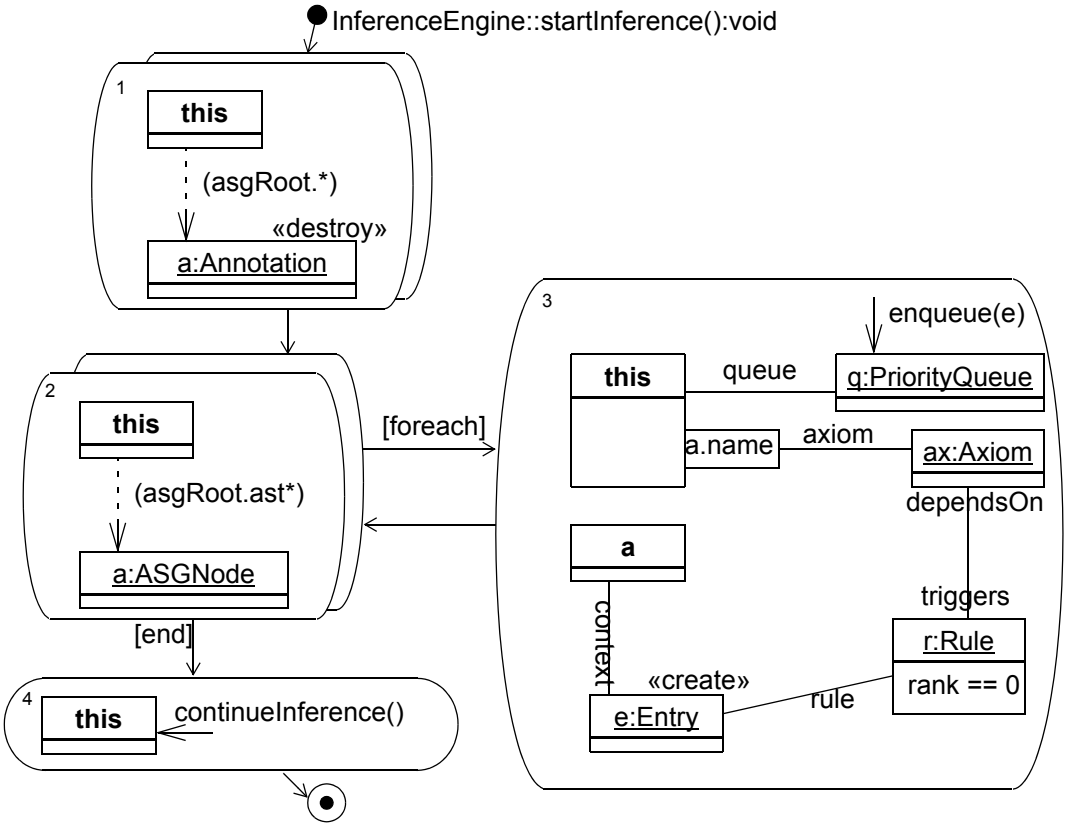


Abbildung 5.10 Inferenz starten

Die Initialisierung der Prioritätsschlange kann unter Umständen einige Zeit dauern, weil alle Blätter des abstrakten Syntaxgraphen als Kontext/Regel Paar in die Prioritätsschlange eingefügt werden und anschließend erst die eigentliche Inferenz gestartet wird. Bei der technischen Umsetzung des Inferenzalgorithmus kann das Füllen der Prioritätsschlange alternativ auch durch einen Thread quasiparallel zum Inferenzalgorithmus erfolgen.

Inferenz fortsetzen

Die eigentliche Inferenz startet mit dem Aufruf der continueInference Methode, siehe Abbildung 5.11, und wird beendet, wenn entweder die Prioritätsschlange leer ist oder die Variable break gesetzt wird. Entsprechend wird im Schritt (1) die Variable break zurückgesetzt und die anschließende Schleifenbedingung daraufhin überprüft, ob noch Elemente in der Prioritätsschlange vorhanden sind.

Der Schleifenrumpf besteht nur aus Schritt (2), der das oberste Element der Prioritätsschlange entnimmt e=dequeue(). Anschließend versucht der Algorithmus im bottom-up Modus, die Regel des Elements e auf den Kontext anzuwenden proceedBottomUp(e).

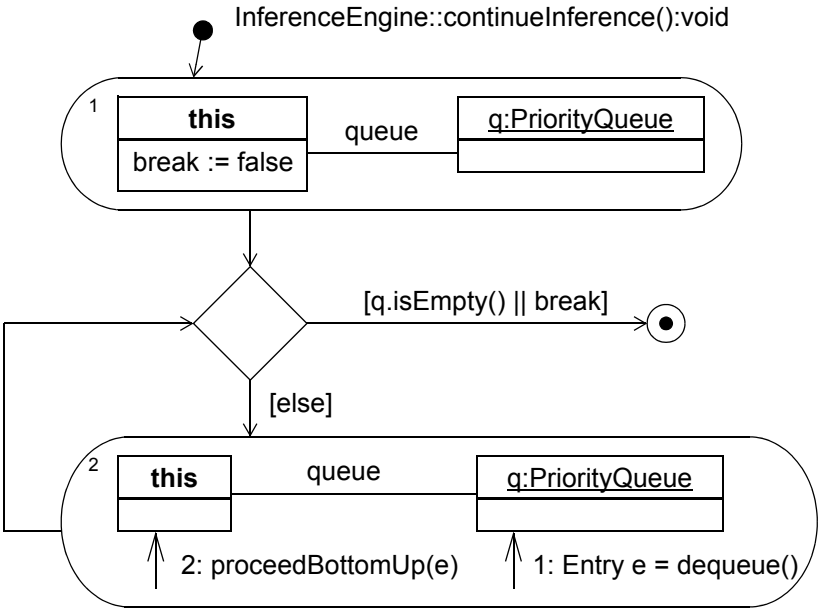


Abbildung 5.11 Inferenz fortsetzen

Anschließend wird überprüft, ob noch Elemente in der Prioritätsschlange vorhanden sind oder der Algorithmus unterbrochen werden soll. Ist dies nicht der Fall, wird das nächste Element aus der Prioritätsschlange entnommen und wiederum im bottom-up Modus versucht die entsprechende Regel auf den Kontext anzuwenden.

Die continueInference Methode ist dabei die Einzige, in der eine Unterbrechung des Algorithmus möglich ist, da die break Variable an keiner anderen Stelle im Algorithmus überprüft wird. Der Algorithmus kann also nur nach einem bottom-up Lauf durch das setzten der break Variable vom Reengineer unterbrochen werden, wie in Abschnitt 5.1 erläutert. Unterbrechungen zu anderen Zeitpunkten würden komplexere Kontrollflüsse erfordern.

Hypothesen validieren

Während des Analyseprozesses hat der Reengineer die Möglichkeit, Hypothesen validieren zu lassen. Hypothesen werden als Annotationen am abstrakten Syntaxgraphen formuliert. Die erzeugten Annotationen müssen dabei konform zum Regelkatalog beziehungsweise zur verfeinerten Schemainformation sein.

Alle Annotationen, die Hypothesen entsprechen, werden in einer Menge gesammelt und können durch die Methode validate der Klasse InferenceEngine auf ihre Gültigkeit hin überprüft werden. Die Überprüfung der Hypothesen entspricht dabei einem top-down Vorgehen des Inferenzalgorithmus.

Die validate Methode in Abbildung 5.12 bekommt die Menge der Hypothesen - erzeugte Annotationen - als Parameter set:AnnotationSet übergeben. Jede Annotation besitzt einen instances-Link zur entsprechenden Regel. Dabei werden in Schritt (1) für jede Annotation in der Menge set in Schritt (2) ein e:Entry Objekte erzeugt, das als Kontext jeweils ein annotiertes Element des abstrakten Syntaxgraphen und die zur Annotation gehörige Regel enthalten. Der Stereotyp «create» bezieht sich dabei sowohl auf das Objekt als auch auf die angrenzenden Links. Da davon ausgegangen werden muss, dass nicht alle Vorgängerregeln der zur Annotation gehörigen Regel ausgeführt worden sind, wird in Schritt (3) versucht das e:Entry Objekt in einem top-down Verfahren zu evaluieren.

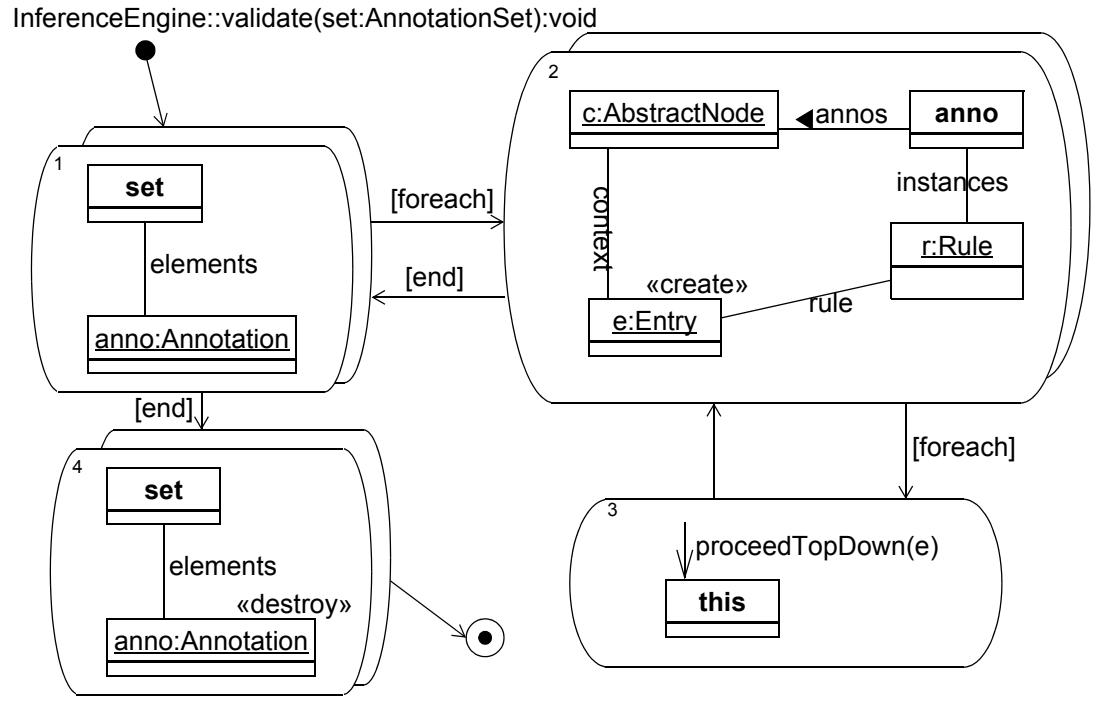


Abbildung 5.12 Hypothesen validieren

Konnte ein top-down Verfahren erfolgreich abgeschlossen werden, so ist eine neue Annotation parallel zur Hypothesenannotation erzeugt worden. Es können also alle Annotationen, die Hypothesen entsprechen und vom Reengineer erzeugt worden sind, in Schritt (4) gelöscht werden.

5.3.5 Bottom-up Modus

Im bottom-up Modus wird versucht, eine Regel auf einen Kontext anzuwenden. Bei erfolgreicher Anwendung der Regel wird eine neue Annotation erzeugt, und es wird versucht, Nachfolgerregeln auf entsprechendem Kontext anzuwenden. Sollte der Fall eintreten, dass die Regel nicht erfolgreich angewendet werden konnte, so kann es prinzipiell zwei unterschiedliche Gründe geben.

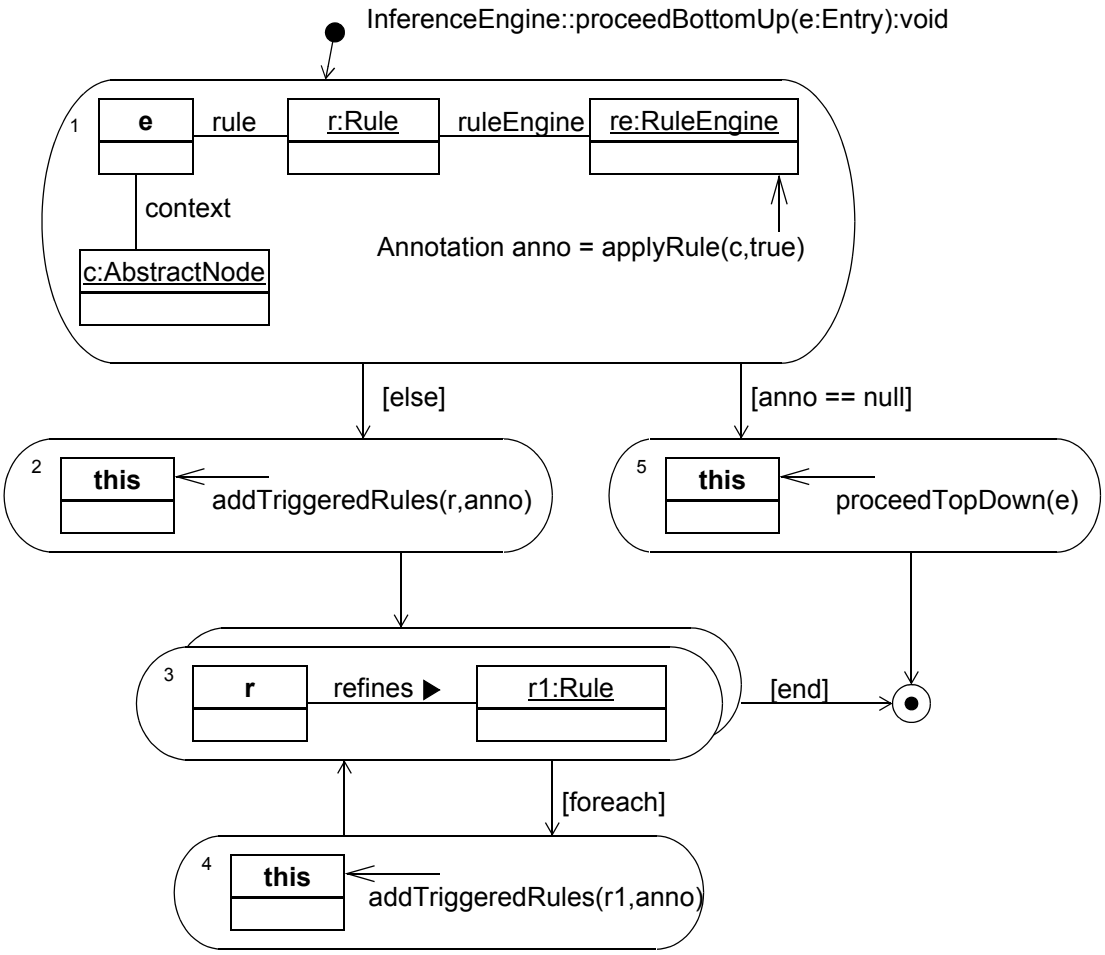


Abbildung 5.13 Bottom-up Inferenz

Erstens könnte es sein, dass die Regel überhaupt nicht auf den Kontext anwendbar ist. In diesem Fall kann eine andere Regel und ein anderer Kontext gewählt werden. Zweitens könnte es sein, dass noch nicht alle Vorgängerregeln auf den Kontext angewendet worden sind, so dass ein Wechsel in den top-down Modus erfolgt, um gegebenenfalls Vorgängerregeln auszuwählen und auf einen zu berechnenden Kontext anzuwenden.

Der Methode proceedBottomUp in Abbildung 5.13 wird als Parameter ein Objekt e:Entry, also ein Regel/Kontext Paar übergeben. Sie versucht in Schritt (1), die Regel im gegebenen Kontext anzuwenden. Die Anwendung erfolgt durch die applyRule Methode der regelspezifischen Klasse, der der Kontext c als Parameter übergeben wird. Der Parameter true besagt, dass sich um eine Anwendung im bottom-up Modus handelt. Der genaue Aufbau der Methode ist im nächsten Abschnitt beschrieben. Ist die Methode erfolgreich ausgeführt worden, so ist eine neue Annotation erzeugt und der temporären Variable anno zugewiesen worden. Im Fall des Misserfolgs enthält die Variable den Wert null.

Bei einer erfolgreichen Ausführung der Methode werden in Schritt (2) die direkten Nachfolgerregeln durch die Methode addTriggeredRules in die Prioritätsschlange eingefügt. Die direkten Nachfolgerregeln sind diejenigen Regeln, die über triggers-Links im Regeltriggergraphen erreichbar sind. Der Kontext ist dabei die neu erzeugte Annotation anno. Aufgrund der Verfeinerungsbeziehungen zwischen Regeln müssen zusätzliche Regeln berücksichtigt werden. Im bottom-up Modus sind dies alle Regeln, die Verallgemeinerungen der aktuellen Regeln darstellen. Dies entspricht der Traversierung des refines-Links entgegen ihrer Leserichtung, siehe Schritt (3). Für alle diese Regeln müssen ihre entsprechenden Nachfolgerregeln mit dem Kontext in die Prioritätsschlange eingefügt werden, Schritt (4). Bezogen auf das Beispiel aus Abbildung 5.1 werden durch die Erzeugung einer MultiLevelGen-Annotation die Regeln Bridge, Strategy und Composite getriggert. Nachdem alle Nachfolgerregeln mit ihrem Kontext in die Prioritätsschlange eingefügt worden sind, ist die Methode beendet.

Sollte in Schritt (1) die Regel auf den Kontext nicht angewendet werden können (anno == null), so wird in den top-down Modus gewechselt und versucht, Vorgängerregeln anzuwenden (proceedTopDown).

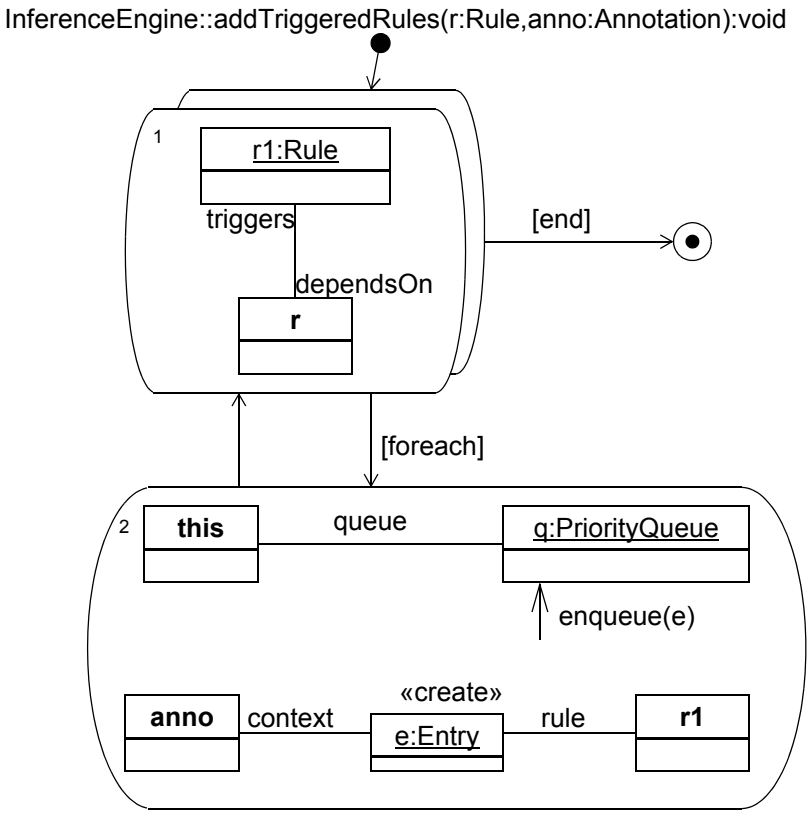


Abbildung 5.14 Neue Regelanwendungsstellen für Bottom-Up

Alternativ zum top-down Modus kann das Regel/Kontext Paar wieder in die Prioritätsschlange eingereiht werden. Dies entspräche dem Vorgehen in klassischen Regelausführungssystemen. Durch die dadurch entstehende schichtweise Abarbeitung des Regeltriggergraphen, können Regeln, die Entwurfsmuster repräsentieren, erst zu einem späten Zeitpunkt angewendet werden.

Neue Regel/Kontext Paare ermitteln

Die Ermittlung neuer Regel/Kontext Paare im bottom-up Modus wird durch die Methode addTriggeredRules realisiert. Sie fügt alle Nachfolgerregeln mit der neu erzeugten Annotation als Regel/Kontext Paar in die Prioritätsschlange ein.

Schritt (1) der Methode sammelt alle Nachfolgerregeln r1:Rule über den triggers-Link der Regel r auf, die als Parameter übergeben worden ist. Für jede Nachfolgerregel r1 wird ein e:Entry Objekt in Schritt (2) erzeugt, die als Kontext die als Parameter übergebene Annotation anno erhält. Das Regel/Kontext Paar wird anschließend in die Prioritätsschlange eingefügt.

5.3.6 Top-down Modus

Im top-down Modus des Inferenzalgorithmus wird versucht, eine Regel, deren Vorgängerregeln im gegebenen Kontext noch nicht angewendet worden sind, anzuwenden. Die Regel, die Auslöser des top-down Modus ist, wird als *ursprüngliche Regel* bezeichnet. Das Ziel ist es, diese ursprüngliche Regel im gegebenen Kontext entweder anzuwenden, oder festzustellen, dass eine Regelanwendung nicht möglich ist. Dabei wird durch die Angabe des Kontextes der Suchraum stark eingeschränkt.

Der Inferenzalgorithmus stellt im top-down Modus sicher, dass alle möglichen Regelanwendungen überprüft werden, um die ursprüngliche Regel, mit der in den top-down Modus gewechselt worden ist, im Kontext auch anwenden zu können. Der durch den Kontext beschränkte Suchraum wird also auf alle alternativen Regelanwendungen hin durchsucht.

Kann eine Regel angewendet werden, so wird die neu erzeugte Annotation mit ihren Nachfolgerregeln als Regel/Kontext Paar zusätzlich mit in die Prioritätsschlange eingefügt. Dadurch muss das Ergebnis nicht mehr im bottom-up Modus überprüft werden.

Sollte der Fall eintreten, dass eine Vorgängerregel im Kontext der ursprünglichen Regel nicht angewendet werden kann und es existieren auch keine alternativen Regel (Verfeinerungen), kann die ursprüngliche Regel auch nicht im gegebenen Kontext angewendet werden. Da alle Alternativen im top-down Modus überprüft worden sind, kann die ursprüngliche Regel auch nicht zu einem späteren Zeitpunkt angewendet werden. Der top-down Modus bricht daher ab.

Im top-down Modus arbeitet der Inferenzalgorithmus mit einem Stack anstelle einer Prioritätsschlange. Dadurch wird sichergestellt, dass keine Zyklen durch quasirekursive Regeln während der Inferenz entstehen, die eine Endlosschleife erzeugen.

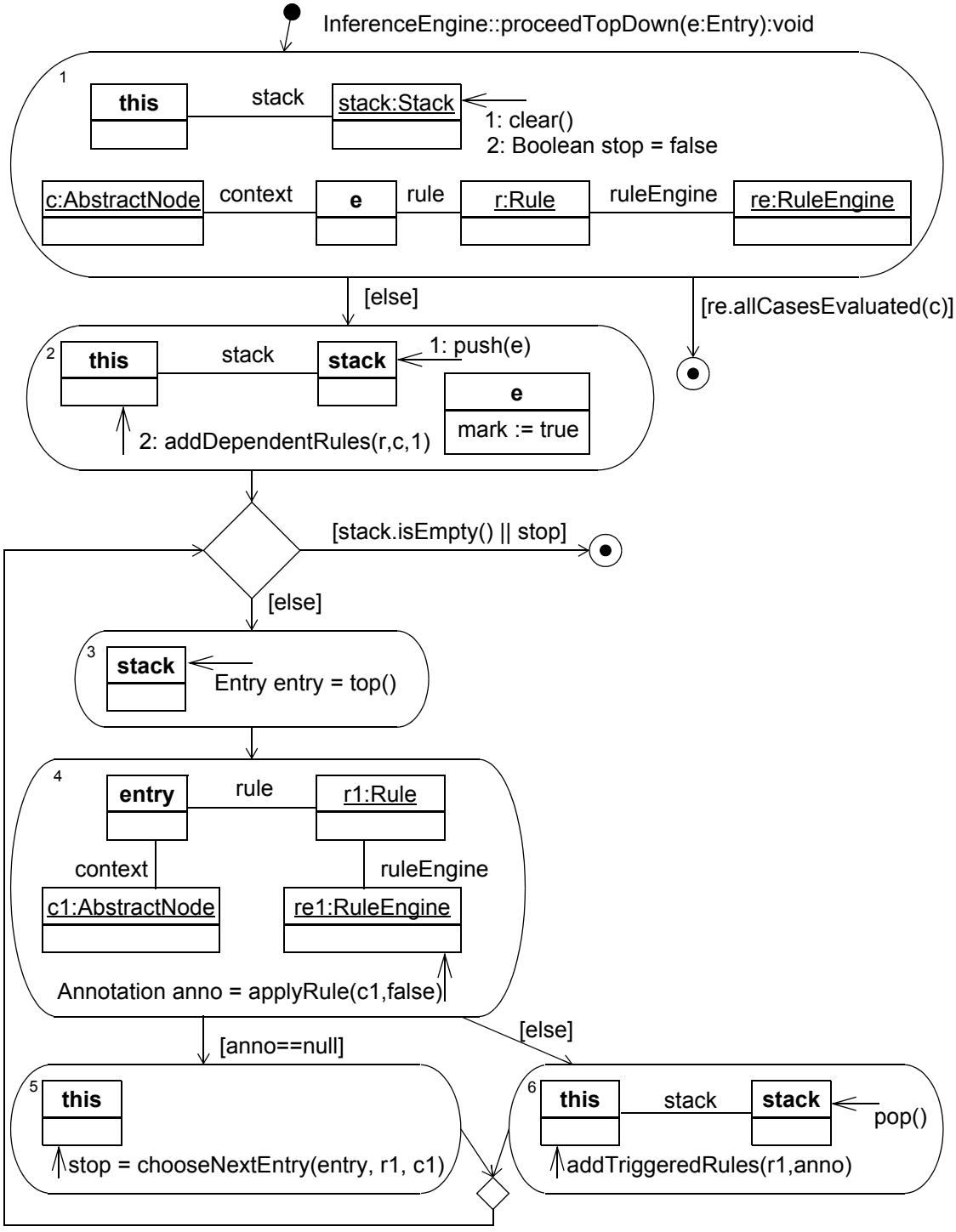


Abbildung 5.15 Top-Down Inferenz

Schritt (1) des Inferenzalgorithmus im top-down Modus, siehe Abbildung 5.15, dient zur Initialisierung des Stacks, der zu Beginn geleert wird (clear). Zusätzlich wird die Variable stop zurückgesetzt, die zum Abbruch des top-down Modus verwendet wird, wenn alle alternativen Regeln auf einem Kontext überprüft worden sind und nicht angewendet werden konnten. Der untere Teil des Story-Pattern wird für die sich anschließende Überprüfung verwendet. Es wird dabei überprüft, ob die Regel r:Rule schon im gegebenen Kontext c:AbstractNode in allen Fällen überprüft worden ist.

Die Überprüfung findet durch die Methode allCasesEvaluated der regelspezifischen Hilfsklasse statt, die im folgenden Abschnitt erläutert wird. Ist die Regel im Kontext bereits überprüft worden, so kann der top-down Modus direkt verlassen werden. Ein solcher Fall kann eintreten, wenn bereits zu einem früheren Zeitpunkt die Anwendung der Regel auf den Kontext fehlgeschlagen ist oder es bereits eine Annotation existiert.

Nach dieser ersten Überprüfung der Notwendigkeit des top-down Modus, wird in Schritt (2) das ursprüngliche Regel/Kontext Paar durch die Methode push auf den Stack gelegt. Da die Regel im bottom-up Modus nicht angewendet werden konnte, werden alle Vorgängerregeln mit einem entsprechenden Kontext durch die Methode addDependentRules auf den Stack gelegt. Dass alle Vorgängerregeln mit entsprechendem Kontext auf den Stack gelegt worden sind, wird im Element e durch das Setzen der mark Variable vermerkt.

Anschließend beginnt der eigentliche Inferenzalgorithmus im top-down Modus mit der Überprüfung, ob noch Elemente auf dem Stack vorhanden sind beziehungsweise der top-down Modus abgebrochen werden kann. Ähnlich wie im bottom-up Modus wird das erste Element des Stacks verwendet - Schritt (3) - und es wird versucht, die Regel auf den Kontext anzuwenden - Schritt (4). Das Regel/Kontext Paar verbleibt dabei auf dem Stack, um die Regel, falls sie auf Vorgängerregeln aufbaut, später erneut anwenden zu können. Die Anwendung der Regel erfolgt wiederum durch die applyRule Methode der regelspezifischen Hilfsklasse.

Sollte die Regel erfolgreich angewendet worden sein - Schritt (6) -, so kann das Regel/Kontext Paar vom Stack entfernt (pop), und die Nachfolgerregeln müssen mit der Annotation als Kontext in die Prioritätsschlange eingefügt werden. Hierzu wird die Methode addTriggeredRules aus dem bottom-up Modus verwendet.

Sollte die Regel nicht angewendet werden können - Schritt (5) -, muss ein neues Regel/Kontext Paar ermittelt werden. Dies erfolgt in der Methode chooseNextEntry. Sowohl nach Schritt (5), wie auch nach Schritt (6) wird ein neues Element des Stacks verwendet und versucht, die enthaltenen Regeln auf den Kontext anzuwenden.

Nächsten Eintrag wählen

Die Methode chooseNextEntry unterscheidet im Wesentlichen zwei Fälle. Im ersten Fall wird eine Regel auf einen Kontext im top-down Modus zum ersten Mal angewendet. Dies kann durch die Variable mark der Klasse Entry überprüft werden. Ist dies der Fall, so wer-

den in Schritt (1) alle Vorgängerregeln mit entsprechendem Kontext als Regel/Kontext Paar auf den Stack gelegt. Hierzu wird wiederum die Methode addDependentRules verwendet. Zusätzlich wird die Variable mark gesetzt. Die Methode ist damit beendet, und der Inferenzalgorithmus kann fortgesetzt werden.

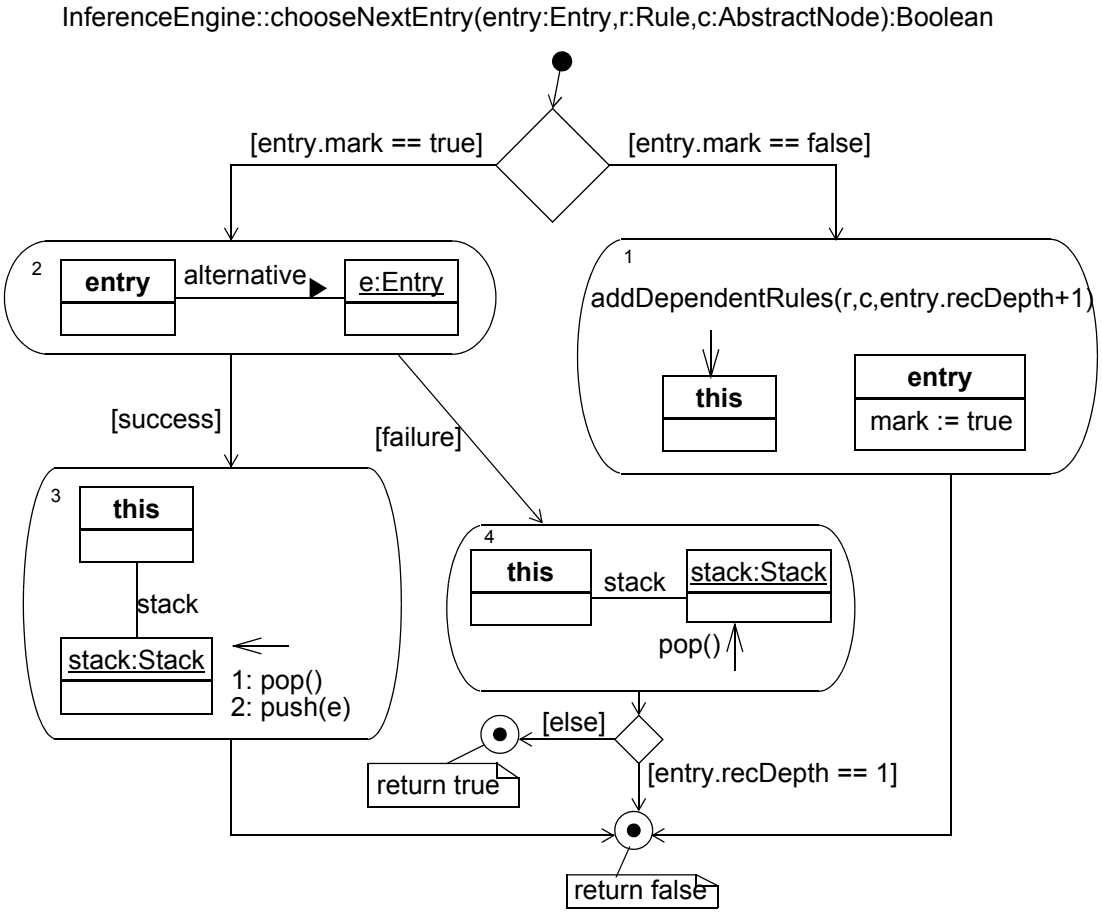


Abbildung 5.16 Hilfsmethode für Top-Down

Der zweite Fall tritt ein, wenn für eine Regel bereits zu einem früheren Zeitpunkt alle Vorgängerregeln mit entsprechendem Kontext auf den Stack gelegt worden sind. Dieser Fall kann auftreten, weil die Elemente auf dem Stack nicht entfernt werden. Es findet dann in Schritt (2) eine Überprüfung statt, ob es alternative Regeln gibt. Alternative Regeln werden durch alternative-Links zwischen Objekten der Klasse Entry repräsentiert, wodurch eine lineare Liste alternativer Regel/Kontext Paare gebildet wird. Dabei liegt nur das erste Element der Liste auf dem Stack. In der Beispielsituation in Abbildung 5.2 ist die 1NAssociation-Regel die Alternative zur Association-Regel. Existieren Alternativen, so wird das oberste Element der linearen Liste entfernt und der Rest wieder auf den Stack gelegt. Gibt es keine Alternative und handelt es sich um eine Regel, von der die ursprünglichen direkt Regel abhängt, so kann der top-down Modus erfolglos abgebrochen werden.

Die chooseNextEntry Methode arbeitet nach einer Tiefensuchstrategie und versucht schnell Regeln mit niedrigem Rang anzuwenden. Anschließend werden aufbauend auf erzeugten Annotationen Regeln auf höheren Rängen versucht anzuwenden. Damit ist das Vorgehen mit einem Branch&Bound Algorithmus vergleichbar, wobei der Algorithmus abgebrochen wird, wenn für einen Teilbaum kein Ergebnis erzeugt werden kann.

Neue Regel/Kontext Paare ermitteln

Die Ermittlung neuer Regel/Kontext Paare ist im top-down Modus in zwei Teile aufgeteilt. Im ersten Teil der Methode addDependentRules wird für jede Vorgängerregel eine Menge möglicher Kontexte ermittelt, und durch die zweite Methode addEntries wird die lineare Liste alternativer Regel/Kontext Paare erzeugt. Alternativen sind dabei nicht nur unterschiedliche Regeln, sondern auch unterschiedliche Kontexte, da als Kontext nur ein Objekt angegeben werden kann.

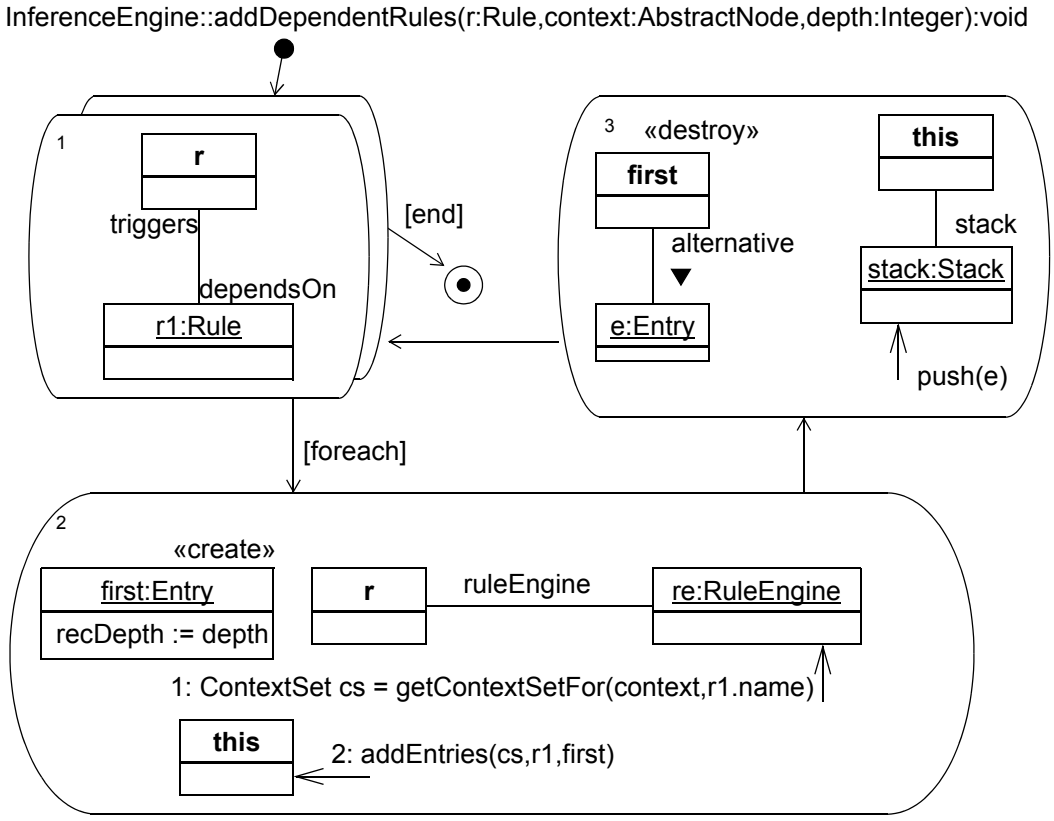


Abbildung 5.17 Neue Regelanwendungsstellen für Top-Down

Schritt (1) der addDependentRules Methode sammelt alle Vorgängerregeln der als Parameter übergebenen Regel auf. Für jede Regel wird eine Menge von Kontexten in Schritt (2) durch die Methode getContextSetFor der regelspezifischen Klasse ermittelt. Der Regelname und der vorhandene Kontext wird dabei als Ausgangspunkt verwendet.

Zusätzlich wird ein first:Entry Objekt der Klasse Entry erzeugt, das als erstes Element der zu erzeugenden linearen Liste verwendet wird. Die lineare Liste selbst wird in der Methode addEntries erzeugt.

Schritt (3) verwendet die erzeugte lineare Liste für die ausgewählte Vorgängerregel, löscht das temporär erzeugte first Objekt und legt das erste Element der Liste auf den Stack. Sollte der Fall eintreten, dass die Liste nur aus dem Element first besteht, so schlägt dieser Schritt fehl. Der Inferenzalgorithmus kann allerdings an dieser Stelle nicht abgebrochen werden, weil es sich um eine Regel handeln könnte, die keine Vorgängerregeln besitzt. Diese Bedingung könnte für eine weitere Optimierung des Algorithmus eingesetzt werden.

Die Methode addEntries erfüllt zwei Aufgaben. Erstens erzeugt sie für alle Elemente der als Parameter übergebenen Kontextmenge ein entsprechendes Regel/Kontext Paar und hängt es an die lineare Liste an. Zweitens werden gegebenenfalls alternative Regeln aufgrund der Verfeinerungsbeziehungen und der Kontextmenge mit in die lineare Liste aufgenommen.

Entsprechend der beiden Aufgaben wird in Schritt (2) für jedes Element der Kontextmenge ein Objekt e:Entry erzeugt, das einem Regel/Kontext Paar entspricht. Der Kontext ist

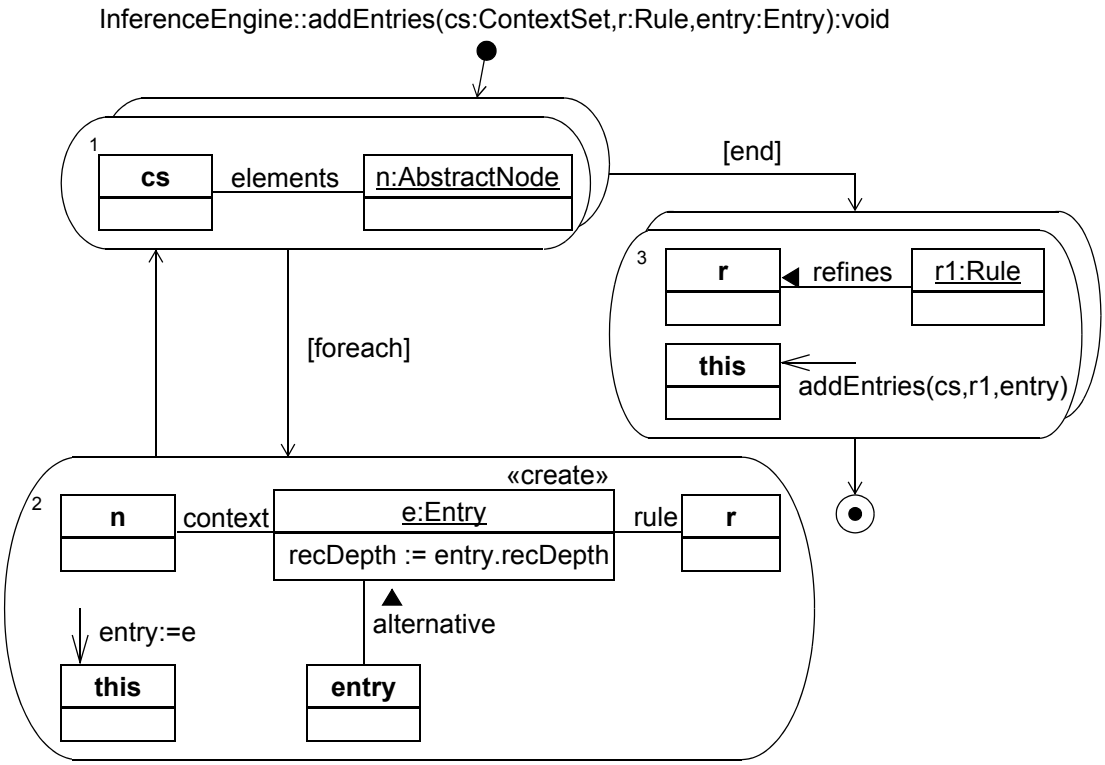


Abbildung 5.18 Hilfsmethode für neue Regelanwendungsstellen

jeweils ein Element - Schritt(1) - der Kontextmenge cs:ContextSet, die als Parameter übergeben wird. Die lineare Liste wird dabei durch einen alternative-Link zum neu erzeugten e:Entry Objekt verlängert. Die Verlängerung der Liste erfolgt durch die Verwendung der entry Variable als Zeiger auf das aktuell letzte Element der Liste (entry := e).

Nachdem alle Elemente der Kontextmenge als Regel/Kontext Paar in die lineare Liste eingefügt worden sind, werden in Schritt (3) alternative Regeln im gleichen Kontext ausgewählt. Alternative Regeln sind alle Regeln, die Verfeinerungen der als Parameter übergebenen Regel sind. Entsprechende Regeln können durch die Traversierung des refines-Links entgegen der Leserichtung ermittelt werden. Durch den rekursiven Aufruf der Methode werden die entsprechenden Regel/Kontext Paare wieder an die Liste angehängt. Dies sichert zu, dass alle möglichen Alternativen in die lineare Liste mit aufgenommen werden.

Im Gegensatz zu bottom-up Regeln, bei denen die refines-Assoziation in Leserichtung traversiert wird, sind im top-down Modus alle Verfeinerungen einer Regel mit zu berücksichtigen. Der Grund dafür ist, dass die Verfeinerungen der Regeln eines Regelkatalogs lediglich strukturelle Verfeinerungen und nicht semantische Verfeinerungen sind. Wären es semantische Verfeinerungen, so müssten diese nicht weiter berücksichtigt werden, weil die allgemeinere Regel in jedem Fall anwendbar wäre.

Die strukturelle Verfeinerung ist ebenfalls der Grund für die zweite Bedingung der Regelränge. Wäre die Bedingung nicht erfüllt, würden immer wieder neue Regel/Kontext Paare erzeugt, die auf Regeln mit höheren Rängen aufbauen. In Kombination mit dem bottom-up Modus könnte der Inferenzalgorithmus in eine Endlosschleife laufen.

5.3.7 Regelspezifische Methoden

Für die Ausführung des Inferenzalgorithmus werden regelspezifische Klassen und dementsprechend Methoden verwendet, die vor der Analyse aus den Regeln generiert werden. Die einmalige Generierung ist dabei gegenüber einer Interpretation der Regeln zeitsparender, da die Methoden während des Inferenzalgorithmus häufig aufgerufen werden.

Regelspezifische Klassen sind durch direkte Vererbungsbeziehungen zur Klasse RuleEngine an Rule Objekte des Regeltriggergraphen gebunden und müssen beim Aufbau des Graphen entsprechend durch ruleEngine-Assoziationen mit einander verbunden werden.

Jede regelspezifische Klasse erhält einen eindeutigen Namen, der sich aus dem Regelnamen mit Präfix “RuleEngine” zusammensetzt. Jede Klasse definiert Methodenrümpfe der abstrakten Klasse RuleEngine für die Methodendeklarationen applyRule, allCasesEvaluated und getContextSetFor.

Im Folgenden wird der Aufbau der einzelnen Methoden anhand der Composite-Regel in Abbildung 5.19 erläutert. Es werden sowohl der generelle Aufbau der Methoden wie auch auf das Beispiel bezogene Teile vorgestellt und diskutiert. Die Composite-Regel ist iden-

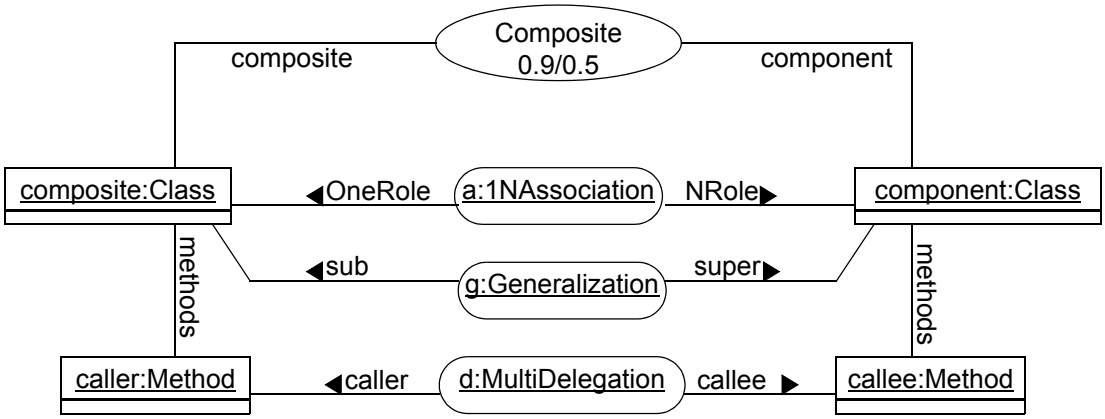


Abbildung 5.19 Composite-Regel Beispiel

tisch mit der im vorherigen Kapitel vorgestellten Regel für die Abbildung auf Story-Pattern.

Methode applyRule

Die Aufgabe dieser Methode ist es, eine Regel in einem gegebenen Kontext anzuwenden. Bei erfolgreicher Anwendung wird eine neue Annotation erzeugt, die mit Annotationen von Vorgängerregeln verbunden wird. Die Methode verwendet dabei die in 5.2 vorgestellten Optimierungen für Regelanwendungen, im Gegensatz zur Abbildung von Regeln auf Story-Pattern, die im vorherigen Kapitel vorgestellt worden ist.

Der generelle Aufbau einer applyRule Methode ist in Abbildung 5.20 dargestellt. Die Inhalte der einzelnen Story-Pattern werden im Anschluss aufgrund von Platzproblemen getrennt erläutert. Die einzelnen Story-Pattern besitzen dabei die gleiche Nummer wie im generellen Aufbau der Methode.

Da die Methode sowohl im bottom-up wie auch im top-down Modus verwendet wird, und der Kontext lediglich aus einem Objekt besteht, wird für alle Annotationen und alle annotierten Objekte einer Regel die abgebildete Struktur von Story-Pattern erzeugt. Dies ist durch die gestrichelte Transition links unten angedeutet.

In Schritt (1) wird zuerst überprüft, ob der als Parameter übergebene Kontext noch überprüft werden muss oder nicht. Dadurch werden doppelte Anwendungen einer Regel vermieden, und der Kontext dient im Folgenden zur Einschränkung der Anwendungsstelle. Schritt (2) versucht, ausgehend vom gegebenen Kontext, eine Anwendungsstelle zu suchen. Modifikationen finden hier nicht statt. Schlägt die Suche fehl, so wird der Kontext als überprüft markiert - Schritt (7). Die Methode ist an dieser Stelle noch nicht beendet, weil es sein kann, dass der Kontext in einer anderen Rolle erneut überprüft werden muss.

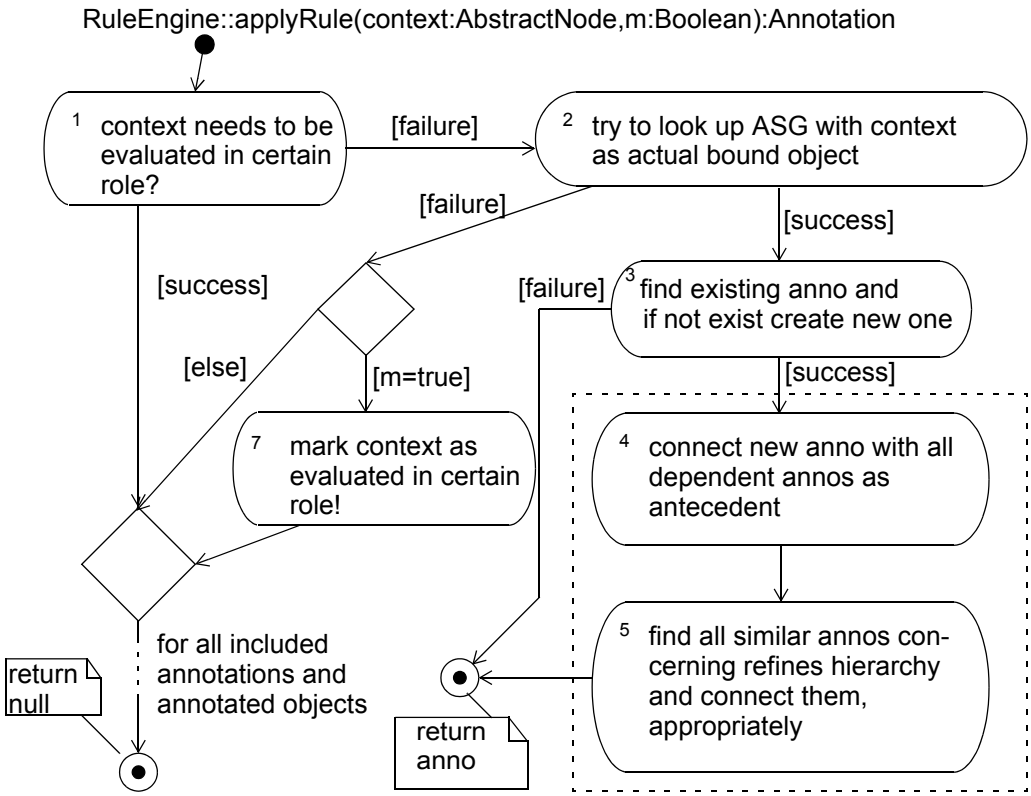


Abbildung 5.20 Prinzipieller Aufbau einer applyRule-Methode

Ist eine Anwendungsstelle gefunden worden, so wird in Schritt (3) überprüft ob schon eine Annotation des Typs existiert und gegebenenfalls eine neue Annotation erzeugt und entsprechende Teile der Anwendungsstelle annotiert. Schritt (4) und Schritt (5), die durch das gestrichelte Rechteck zusammengefasst werden, dienen dazu, die neue Annotation mit Annotationen von Vorgängerregeln und Annotationen alternativer Regeln zu verbinden. Die Trennung in zwei Schritte hat lediglich technische Gründe. Die Links, die in diesen beiden Schritten erzeugt werden, dienen zur Berechnung des Präzisionswerts der Annotationen, die im Anschluss an die Inferenz stattfindet. Die neu erzeugte Annotation wird anschließend an die aufrufende Methode zurückgegeben.

Für die Composite-Regel aus Abbildung 5.19 sind insgesamt fünf Kontrollflussteile wie in Abbildung 5.20 notwendig, weil die Regel die beiden Klassen c1:Class und c2:Class annotiert und auf Annotationen der Regeln Generalization, 1NAssociation und MultiDelegation aufbaut. Exemplarisch wird im Folgenden der Aufbau der Schritte für eine Generalization Annotation als Kontext erläutert. Für die restlichen vier Kontexte sind die Schritte entsprechend aufgebaut.

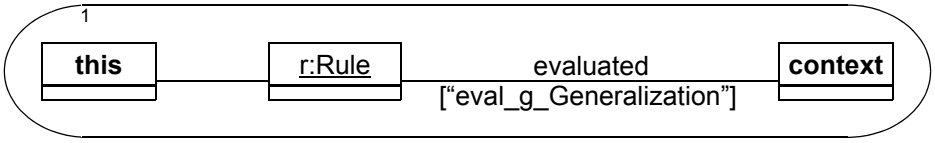


Abbildung 5.21 Composite-Regel: Evaluierung notwendig?

In Schritt (1) der applyRule Methode mit Kontext Generalization wird zuerst überprüft, ob der Kontext bereits überprüft worden ist. Sollte dies der Fall sein, so existiert ein Link zwischen dem r:Rule Objekt und dem context Objekt unter dem Schlüssel "eval_g_Generalization". Es handelt sich dabei um einen eindeutigen Schlüssel, der aus der Regel generiert wird. Ein Schlüssel wird für alle Kontexte generiert, die von der Regel nicht annotiert werden. Für alle annotierten Objekte wird der Name der Kante verwendet. Für die Composite-Regel existieren die Schlüssel "component", "composite", "eval_g_Generalization", "eval_a_1NAssociation" und "eval_d_MultiDelegation". Diese Schlüssel werden nur noch von anderen regelspezifischen Methoden verwendet, wodurch die Schlüssel gegenüber dem restlichen Inferenzalgorithmus gekapselt werden.

Schritt (2) beinhaltet die Suche der Anwendungsstelle ausgehend vom gegebenen Kontext. Das generierte Story-Pattern wird aus der Regel gemäß der im vorherigen Kapitel beschriebenen Abbildung erzeugt, wobei es keine Modifikationen enthält. Außerdem wird das context Objekt als Markierung der Anwendungsstelle in das Story-Pattern mit eingebaut. Der g=(Generalization)context Ausdruck bedeutet, dass ein Typkonvertierung des context Objekts auf ein Objekt vom Typ Generalization stattfindet. Ausgehend von diesem Objekt wird der restliche Teilgraph zu binden versucht. Als Grundlage wird die qualifizierte annos-Assoziation der verfeinerten Schemainformation des abstrakten Syntaxgraphen mit entsprechenden Schlüsseln traversiert. Die Schlüssel entsprechen den Namen der Kanten in der Regel, vergleiche vorheriges Kapitel.

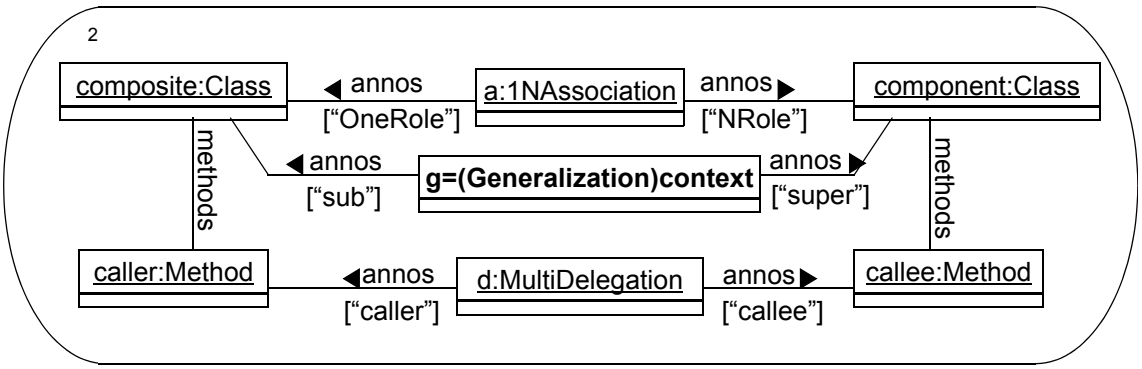


Abbildung 5.22 Composite-Regel: Suche Anwendungsstelle von Kontext aus

Durch die Verwendung des context Objekts zur Markierung der Anwendungsstelle kann auf die optimierte Regelanwendung aus 5.2 zurückgegriffen werden. Es wäre auch möglich gewesen, hier die konstruierte Regel aus dem vorherigen Kapitel zu verwenden, was aber die Laufzeit der Teilgraphensuche erhöht hätte.

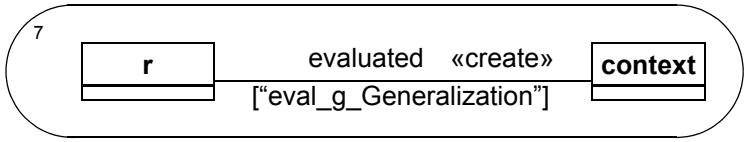


Abbildung 5.23 Composite-Regel: Kontext als evaluiert kennzeichnen

Konnte keine Anwendungsstelle gefunden werden, so wird das context Objekt in Schritt (7) als überprüft markiert wenn sich der Inferenzalgorithmus im bottom-up Modus befindet. Im top-down Modus ist dies nicht möglich, weil es sein kann, dass die Suche der Anwendungsstelle in Schritt (2) durch noch nicht vorhandene Annotationen, die erst im weiteren Verlauf des top-down Modus erzeugt werden, gescheitert ist. Die Markierung erfolgt durch die Erzeugung eines evaluated-Links zwischen der Regel und dem Kontext mit dem Schlüssel "eval_g_Generalization".

Nachfolgend wird der Kontext noch in allen weiteren Rollen überprüft, die allerdings alle fehlschlagen, da eine Generalization Annotation nur in einer Rolle in der Composite-Regel vorkommt. Im Falle eines Objekts vom Typ Class existieren zwei Rollen, in der die Klasse überprüft werden muss. Da die Kontrollflussteile aus Abbildung 5.20 sequentiell für jedes Objekt verbunden werden, ist die Überprüfung der beiden Rollen gewährleistet.

Bei erfolgreicher Suche einer Anwendungsstelle wird in Schritt (3) eine neue Annotation newAnno:Composite erzeugt, und die beiden Objekte component und composite, die die entsprechenden Klassen repräsentieren, annotiert. Die Annotation erfolgt durch die Erzeugung von annos-Links mit den Schlüsseln "component" und "composite".

Die Annotation erfolgt allerdings nur, wenn nicht schon eine entsprechende Annotation existiert, was durch das Negieren des Objekts mit Hilfe des Kreuzes geprüft wird. Eine Annotation kann möglicherweise schon erzeugt worden sein, wenn die Regel schon einmal auf den Context angewendet worden ist und diese Anwendung erfolgreich war.

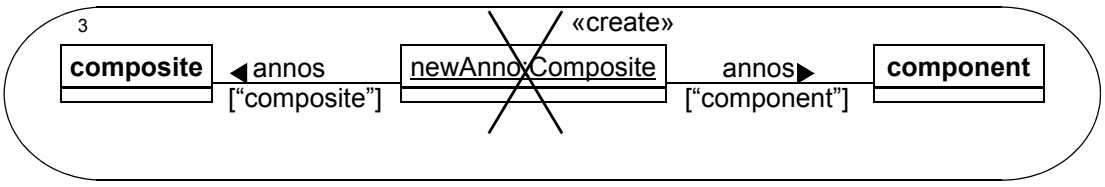


Abbildung 5.24 Composite-Regel: Annotation erzeugen

Allgemein ist die Negation in Kombination mit einem Erzeugen des Objekts ist eine abkürzende Schreibweise für einen Kontrollfluss, in dem erst die Existenz des Objekts und deren Links überprüft wird und bei der Nichtexistenz erzeugt wird.

In diesem Schritt werden dabei die Objekte nicht erneut gesucht, sondern es werden lediglich das neue Annotationsobjekt und Links erzeugt. Die anderen Objekte werden aus dem vorherigen Story-Pattern übernommen. In [Zün01] wird dies als *Logical-Successor-Story-Pattern* bezeichnet, weil es als zu suchenden Teilgraphen den Graphen enthält, der sich aus der Anwendung des vorherigen Story-Pattern ergibt.

Schritt (4) und Schritt (5) verwenden Ausschnitte eines Logical-Successor-Story-Pattern für die Erzeugung von Links des neu erzeugten Annotationsobjekts newAnno mit anderen Annotationen. Die erzeugten Links in den beiden Schritten werden für die anschließende Berechnung des Präzisionswerts der Annotationen verwendet.

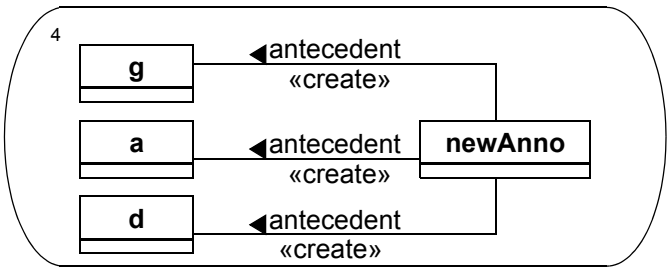


Abbildung 5.25 Composite-Regel: Abhängige Annotation markieren

Schritt (4) erzeugt antecedent-Links zwischen der neu erzeugten Annotation und den Annotationen, die in der Anwendungsstelle vorkommen.

Die Beziehung zwischen antecedent-Links und triggers/dependsOn-Links kann mit der Beziehung zwischen Links und Assoziationen in Klassendiagrammen verglichen werden. Existieren im Regeltriggergraphen triggers/dependsOn-Links zwischen Regeln, so existieren entsprechende antecedent-Links zwischen erzeugten Annotationen der Regeln.

Schritt (5) der applyRule Methode überprüft, ob aufgrund der Verfeinerungsbeziehungen zwischen den Regeln bereits andere Annotationen existieren und verbindet diese mit similar-Links. Der Schritt kann dabei aus mehreren Teilschritten bestehen, da die Verfeinerungsbeziehung sowohl in Leserichtung, als auch entgegen der Leserichtung der refines-Assoziation überprüft werden muss.

Der erste Teilschritt (5a) behandelt dabei alle Verfeinerungsbeziehungen, die entgegen der Leserichtung verlaufen. Allgemein folgt aus den Konsistenzbedingungen eines Regelkatalogs, dass alle Verfeinerungen einer Regel mindestens alle diejenigen Objekte annotieren, die die verfeinerte Regel annotiert. Durch die Polymorphieeigenschaft des Graphmodells werden im ersten Teilschritt alle Annotationsobjekte anno:Composite von Verfeinerungen der Composite-Regel aufgesammelt und mit einem similar-Links entge-

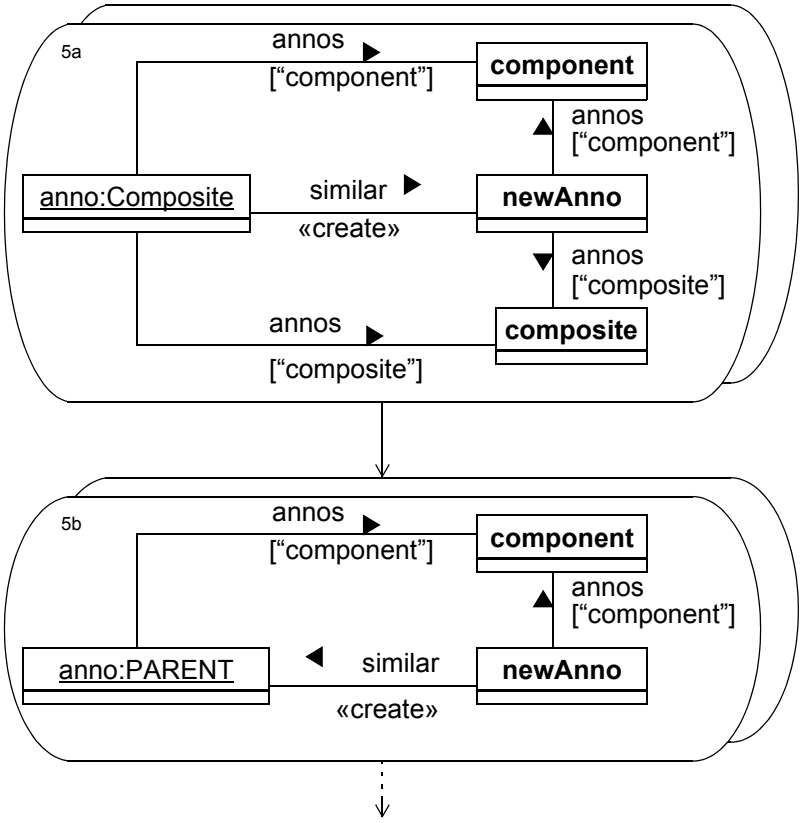


Abbildung 5.26 Composite-Regel: Ähnliche Annotationen markieren

gen der Leserichtung ausgehend von der neuen Annotation newAnno verbunden. Die Isomorphieeigenschaft der Teilgraphensuche eines Story-Pattern verhindert dabei, dass ein similar-Link zur neuen Annotation newAnno selbst erzeugt wird.

Für Verfeinerungsbeziehungen von Regeln in Leserichtung gilt die Konsistenzbedingung eines Regelkatalogs bezüglich der annotierten Objekte ebenfalls. Welche Objekte annotiert werden, bestimmt aber die Regel, die verfeinert worden ist. Es kann vorkommen, dass für jede Verfeinerungsbeziehung ein eigenes Story-Pattern generiert werden muss - Teilschritt (5b). Dabei muss die Klasse der aufzusammelnden Annotationen PARENT entsprechend angepasst werden und nur diejenigen Objekte aufgeführt werden, die die PARENT Regel annotiert. Dies ist angedeutet durch das Fehlen der als "composite" annotierten Objekte. In diesem Fall werden similar-Links in Leserichtung ausgehend von der neu erzeugten Annotation erzeugt. Die Beziehung zwischen refines und similar-Links kann also ebenfalls mit der Beziehung zwischen Assoziationen und Links verglichen werden.

Im Beispiel der Composite-Regel ist Schritt (5) nicht notwendig, da Composite keine Verfeinerung noch eine Verallgemeinerung hat. Bei einer MultiReference Regel müssen al-

lerdings die Teilschritte (5a) und (5b) generiert werden, wobei PARENT durch Reference ersetzt wird, und die Schlüssel angepasst werden müssen.

Methode allCasesEvaluated

Die Methode allCasesEvaluated dient in erster Linie dazu, zu überprüfen, ob noch eine Möglichkeit besteht, eine Regel in dem als Parameter übergebenen Kontext anzuwenden. Diese Möglichkeit ist gegeben, wenn zum Kontext noch nicht alle möglichen evaluated-Links bestehen. Zusätzlich dient die Methode dazu, die Schlüssel, unter denen Kontexte in der evaluated-Assoziation eingetragen werden, zu kapseln.

Ein Ausschnitt des Aufbaus der Methode ist in Abbildung 5.27 dargestellt. Der Ausschnitt muss dabei, wie bei der applyRule Methode, für alle Annotationen und annotierten Objekte erzeugt werden.

Schritt (1) der Methode dient zur Initialisierung - dem Binden des Objekts r:Rule, das für die nachfolgenden Überprüfungen verwendet wird. Dieser Schritt wird also lediglich einmal generiert.

Für alle Annotationen und annotierten Objekte wird als nächstes der Typ des Kontextobjekts bestimmt. Im Beispiel erfolgt dies durch context instanceOf Generalization. Anschließend wird in Schritt (2) überprüft, ob der Kontext bereits überprüft worden ist. Es

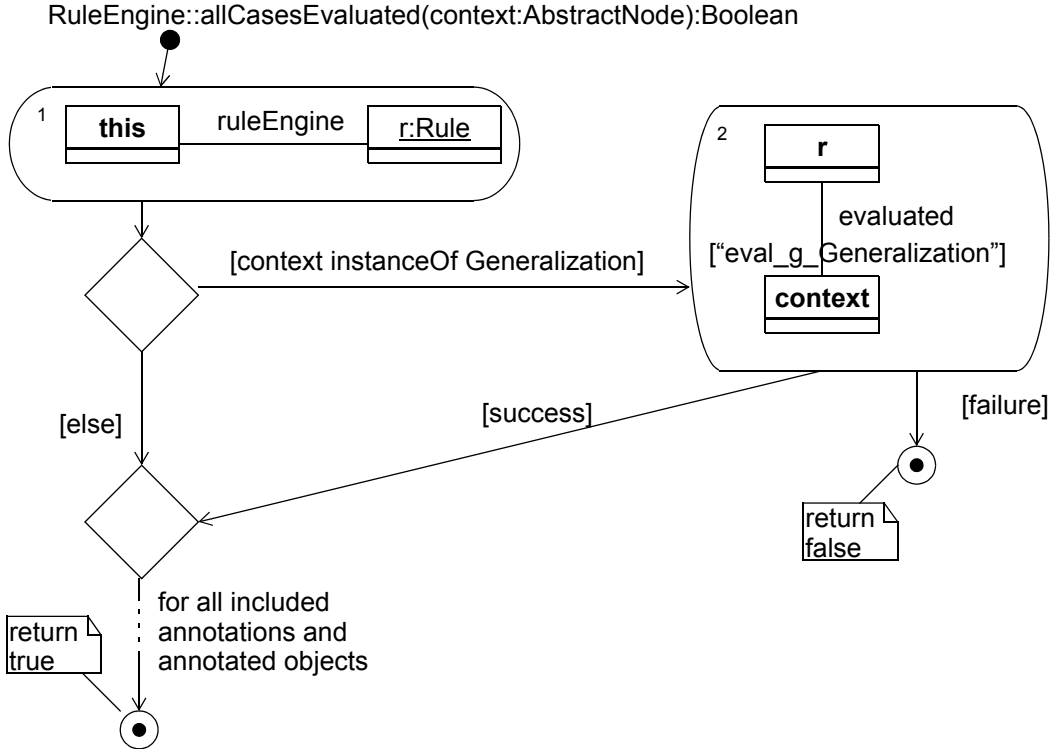


Abbildung 5.27 Composite-Regel: allCasesEvaluated-Methode (Ausschnitt)

muss ein evaluated-Link unter entsprechendem Schlüssel existieren. Ist dies der Fall, so müssen gegebenenfalls weitere Rollen für den Kontext überprüft werden. Ist der Kontext noch nicht in der Rolle überprüft worden, so kann die Methode mit dem Rückgabeparameter false beendet werden.

Die Typüberprüfung und die Existenzüberprüfung der evaluated-Assoziation kann nicht wie in der applyRule Methode gleichzeitig innerhalb des Schrittes (2) erfolgen, weil die Typ-Überprüfung oder die Existenzprüfung fehlschlagen könnte. Relevant für das Ergebnis der Methode ist allerdings nur die Existenz eines evaluated-Links.

Für die Composite-Regel wird also für die fünf Schlüssel "component", "composite", "eval_g_Generalization", "eval_a_1NAssociation" und "eval_d_MultiDelegation" jeweils eine eigene Überprüfung generiert. Dabei wird im Falle der Schlüssel "component" und "composite" jeweils auf den Typ Class geprüft.

Methode getContextSetFor

Die getContextSetFor Methode wird im top-down Modus verwendet, um Kontexte von anzuwendenden Vorgängerregeln zu bestimmen. Im Gegensatz zum bottom-up Modus, indem eine neu erzeugte Annotation als Kontext für Nachfolgerregeln verwendet wird, muss der Kontext im top-down Verfahren für die Vorgängerregeln individuell für jede Regel ermittelt werden. Da es in der Regel mehr als ein Kontextobjekt für eine Vorgängerregel gibt, werden alle Kontextobjekte aufgesammelt und als Menge cs:ContextSet zurückgeliefert, die in Schritt (1) neu erzeugt wird.

Kontextobjekte werden jeweils für eine Vorgängerregel ermittelt. Die Methode enthält daher für alle Vorgängerregeln eine Fallunterscheidung, wobei die konkrete Vorgängerregel, für die ein Kontext ermittelt werden soll, durch ihren Namen identifiziert und als Parameter übergeben wird. Im Beispiel der Composite-Regel sind dies die Regeln MultiDelegation, Generalization und 1NAssociation. Abbildung 5.28 beinhaltet die Ermittlung der Kontextobjekte für die MultiDelegation-Regel.

Die Ermittlung der Kontextobjekte für eine bestimmte Vorgängerregel erfolgt durch die Konstruktion von Story-Pattern. Für jede Annotation einer Regel, die durch eine Vorgängerregel erzeugt werden kann, werden alle Wege von den annotierten Objekten der Regel zu den annotierten Objekten der Vorgängerregel ermittelt und als Kontextobjekt in einer Menge gespeichert.

Zum Beispiel wird für die Kontextbestimmung für die MultiDelegation Vorgängerregel der Composite-Regel ausgehend vom composite:Class oder component:Class Objekt alle Objekte m:Method ermittelt, die als Kontextobjekte in die Menge aufgenommen werden. Das Aufsammeln erfolgt in Schritt (2), wobei es einen direkten Weg von einer der beiden Klassen component oder composite zu den Methoden gibt. Das Hinzufügen eines Kontextobjekts zur Menge ist in Schritt (3) realisiert.

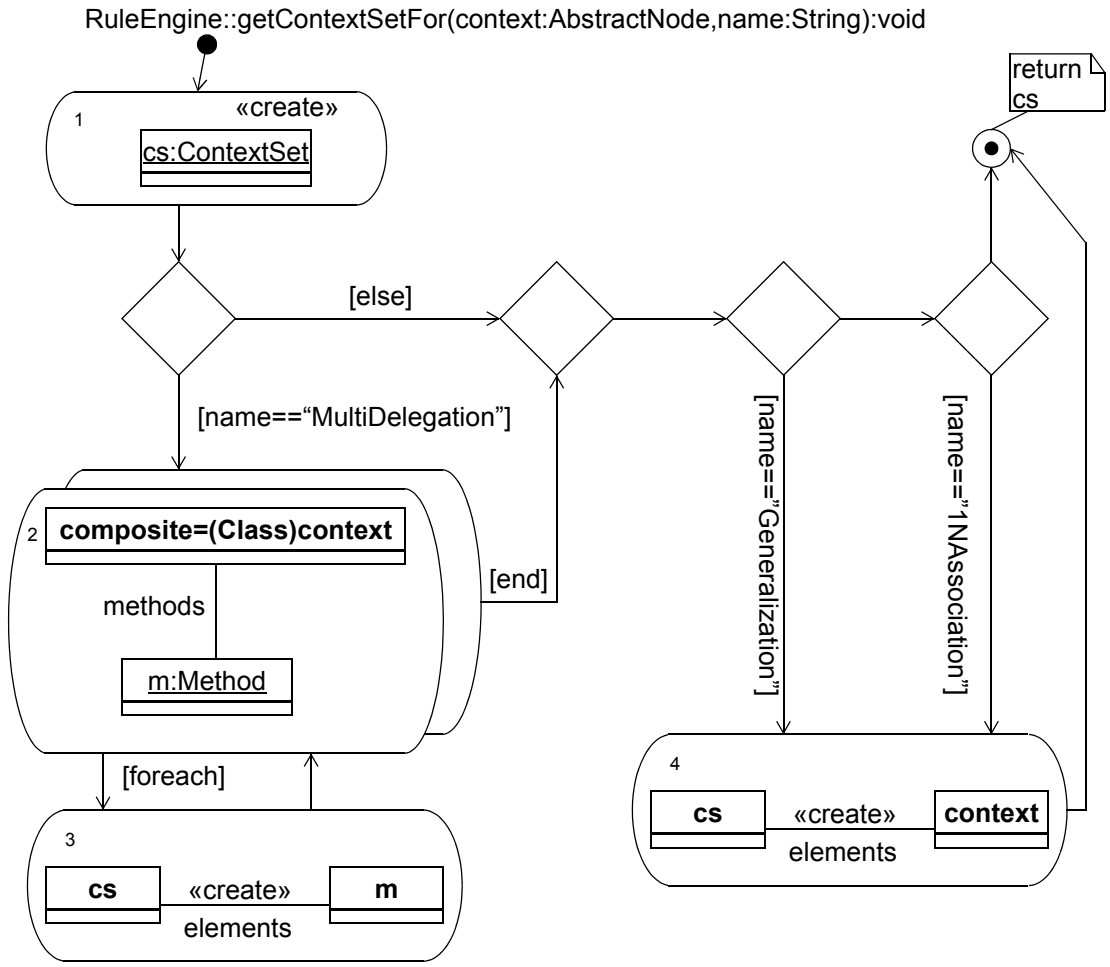


Abbildung 5.28 Composite-Regel: getContextSetFor-Methode (Ausschnitt)

Prinzipiell müssten nach Schritt (3) weitere mögliche Wege zu Kontextobjekten generiert werden. Im Beispiel der Composite-Regel ist dies allerdings nicht notwendig. Der verbliebene Weg, ausgehend vom component Objekt, ist mit dem obigen identisch und kann daher entfallen. Ebenso können die Wege vom composite Objekt über eines der Annotationsobjekte entfallen, weil die unterschiedlichen Rollen, in denen ein Kontextobjekt bei einer späteren Regelanwendung verwendet wird, keinen Einfluss haben.

Die Kontextbestimmung für die Vorgängerregeln Generalization und 1NAssociation besteht lediglich aus dem Hinzufügen des context Objekts zur Menge. Der Grund dafür ist, dass der übergebene Kontext mit dem Kontext der Vorgängerregeln übereinstimmt. Aus Platzgründen ist Schritt (4) für die Bestimmung der Kontextobjekte für beide Vorgängerregeln wieder verwendet worden.

Ein Problem, das durch die vorher beschriebene Generierung nicht optimal gelöst wird, ist, wenn eine Regel mehrere gleiche Vorgängerregeln besitzt. Durch die beschriebe Ge-

nerierung werden dann für alle gleichen Vorgängerregeln Kontextobjekte zur Menge hinzugefügt. Die Menge der Kontextobjekte enthält gegebenenfalls dadurch mehr Objekte als notwendig. Das Problem kann im Inferenzalgorithmus durch kompliziertere Kontrollstrukturen in den einzelnen Methoden sowie weitere Datenstrukturen gelöst werden. Allerdings hat sich in der Praxis gezeigt, dass der zusätzliche Aufwand, der durch die vorgestellte Generierung erzeugt wird, vernachlässigbar ist, da Regeln mit mehreren identischen Vorgängerregeln selten in einem Regelkatalog vorkommen.

5.3.8 Beispiel

Als Beispiel für das Ergebnis des Inferenzalgorithmus und für die anschließende Berechnung der Präzisionswerte der Ergebnisse ist die Abstract-Windowing-Toolkit-Bibliothek (AWT) des Java Development Kits (JDK) analysiert worden. Dabei ist der Regelkatalog aus Abbildung 5.1 verwendet worden. Die Regeln des Katalogs sind schon teilweise für die Analyse der Bibliothek angepasst worden.

Allgemein ist das Ergebnis des Inferenzalgorithmus ein annotierter abstrakter Syntaxgraph. Abbildung 5.29 zeigt einen Ausschnitt des annotierten abstrakten Syntaxgraphen nach der Ausführung des Inferenzalgorithmus auf dem Quelltext der AWT-Bibliothek als Objektdiagramm. Der grau unterlegte Teil repräsentiert den gesamten annotierten abstrakten Syntaxgraphen, wobei die Objekte der Component-Klasse und der Composite-Klasse der Bibliothek explizit dargestellt sind. Beide Klassen enthalten neben einer Reihe weiterer Methoden jeweils eine paint-Methode - repräsentiert durch die beiden Method-Objekte. Außerhalb der grauen Hinterlegung ist eine Teilmenge der erzeugten Annotationsobjekte dargestellt. Die Teilmenge besteht aus Annotationsobjekten, die aufgrund der transitiven Abhängigkeit der Composite-Regel erzeugt worden sind. Aus Platzmangel sind nicht alle Annotationsobjekte der transitiven Abhängigkeit mit ihren Links zu anderen Objekten dargestellt. Entsprechende Annotationsobjekte sind gestrichelt dargestellt. Abbildung 5.29 beinhaltet zusätzlich zu den Objekten Links. Dabei werden annos-Links einfach durch ihren Namen und die antecedent-Links nur durch Pfeile für eine bessere Übersicht dargestellt. Die beiden kleinen Kreise dienen ebenfalls der besseren Übersicht und sind eine Abkürzung.

Die Composite-Regel aus Abbildung 5.3 annotiert die beiden Objekte, die die Klasse Component und Containter der AWT-Bibliothek repräsentiert. Entsprechend der Abhängigkeit der Composite-Regel existieren antecedent-Links zu den Annotationsobjekten MultiDelegation, Generalization und 1NAssoziation. Diese Annotationsobjekte besitzen wiederum annos- und antecedent-Links. Zwischen den Annotationsobjekten 1NAssociation und 1NArrayAssoc existiert ein similar-Link, weil zwischen den zugehörigen Musterinstanzregeln eine Verfeinerungsbeziehung besteht, siehe Abbildung 5.1. Zusätzlich zu einer existierenden Verfeinerungsbeziehung ist es für die Erzeugung eines similar-Links notwendig, dass die beiden Annotationen dieselben Objekte annotieren, was im Beispiel der Fall ist.

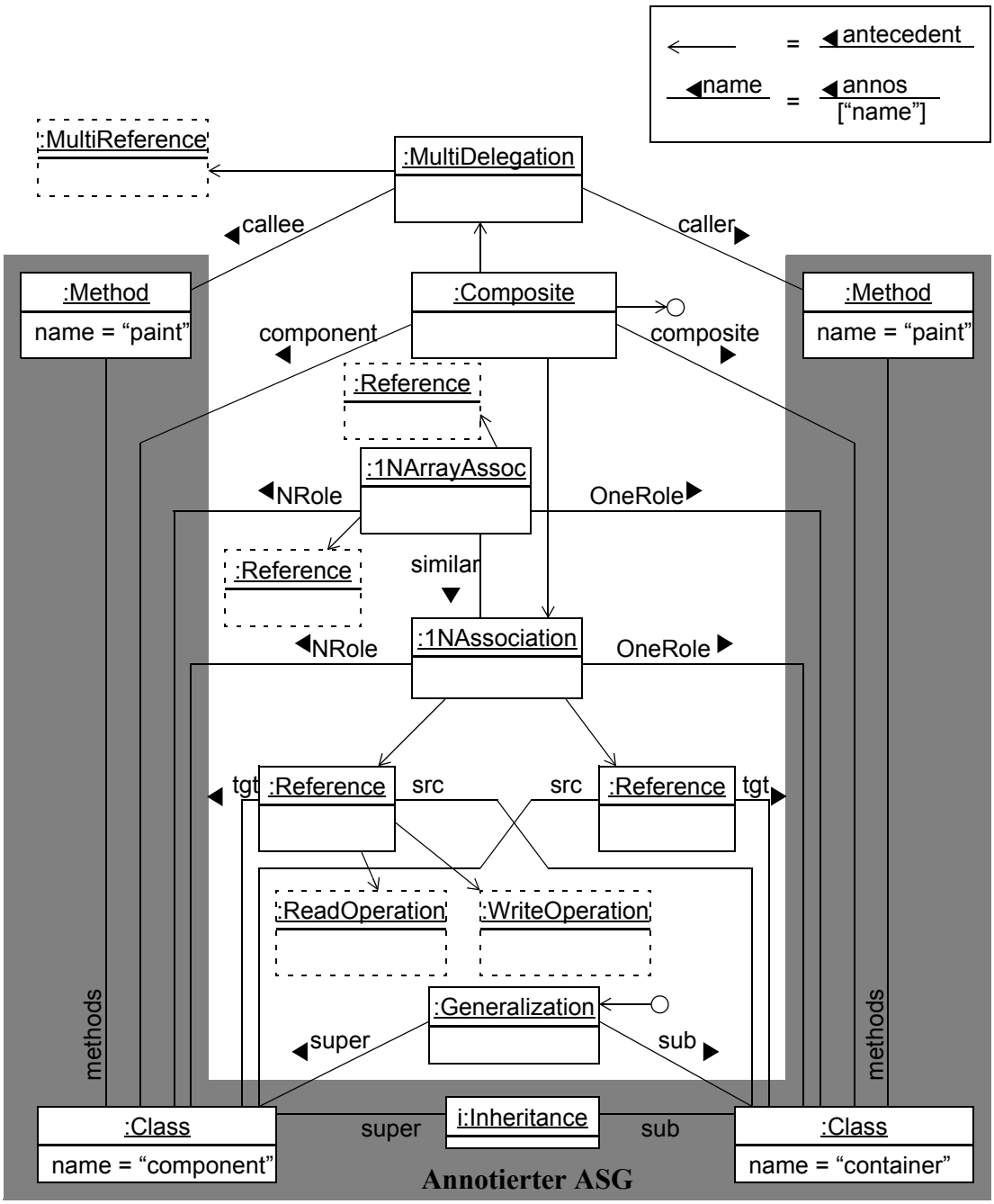


Abbildung 5.29 Beispiel für Objektstrukturausschnitt nach Inferenzalgorithmus

5.4 Berechnung der Präzisionswerte

Die Berechnung der Präzisionswerte der Ergebnisse - Annotationen - findet nach dem Inferenzalgorithmus beziehungsweise nach einer Unterbrechung des Algorithmus statt. Eine Berechnung während des Inferenzalgorithmus ist zwar ebenfalls möglich, allerdings erfordern Korrekturen an Präzisionswerten von Annotationen, die durch den Reengineer vorgenommen worden sind, eine anschließende Neuberechnung der Präzisionswerte. Daher findet die Berechnung getrennt vom Inferenzalgorithmus statt.

Grundlage für die Berechnung der Präzisionswerte von Annotationen bilden die Genauigkeits- und die Schwellwerte der einzelnen Musterinstanzregeln. Die vom Inferenzalgorithmus erzeugten Annotationen mit ihren Abhängigkeitsbeziehungen können dabei als Petrinetz aufgefasst werden. Die Berechnung der Präzisionswerte erfolgt daher durch ein Fuzzy-Petrinetz [KM96]. Das Fuzzy-Petrinetz (FPN) ist ein Stellen-Transitionsnetz, wobei erzeugte Annotationen den Stellen entsprechen, und die antecedent-Links zwischen den Annotationen mit Transitionen korrespondieren. Jede Stelle erhält ein so genanntes fuzzy-belief-marking, das einen Wert im Bereich [0..1] annehmen kann. Das fuzzy-belief-marking entspricht dem Präzisionswert einer Annotation. Definition 5.4 beschreibt das hier verwendete Fuzzy-Petrinetz. Seine Konstruktion aus den (Zwischen-)Ergebnissen des Inferenzalgorithmus und seine Auswertung wird in den folgenden Abschnitten beschrieben.

Definition 5.4: Fuzzy-Petrinetz

Ein Fuzzy-Petrinetz ist ein Tupel FPN := (S, T, F, bf, th, ubm, cbm, fbm) mit

- *S ist eine endliche Menge von Stellen*
- *T ist eine endliche Menge von Transitionen*
- $F \subset (S \times T) \cup (T \times S)$ *ist die Menge der Kanten des Netzes*
- $bf : S \rightarrow [0..1]$ *ordnet jeder Stelle einen Belief zu*
- $th : S \rightarrow [0..1]$ *ordnet jeder Stelle einen Threshold zu*
- $ubm : S \rightarrow [0..1]$ *ist partiell und ordnet Stellen einen Fuzzywert des Reengineer zu*
- $cbm : S \rightarrow [0..1]$ *ordnet jeder Stelle ein berechnetes fuzzy-belief-marking zu.*
- $fbm : S \rightarrow [0..1]$ *ordnet jeder Stelle ein fuzzy-belief-marking zu, wobei gilt:*
  $\forall s \in S$ *ist* $fbm(s) = ubm(s)$ *falls definiert, sonst* $fbm(s) = cbm(s)$

5.4.1 Erzeugung des Fuzzy-Petrinetzes

Das Fuzzy-Petrinetz für die Berechnung der Präzisionswerte wird aus der Objektstruktur, die durch den Inferenzalgorithmus erzeugt worden ist, konstruiert. Die Menge der Stellen *S* entspricht den erzeugten Annotationen des Inferenzalgorithmus, und die Transitionsmenge *T* entspricht den existierenden antecedent-Links zwischen den Annotationen.

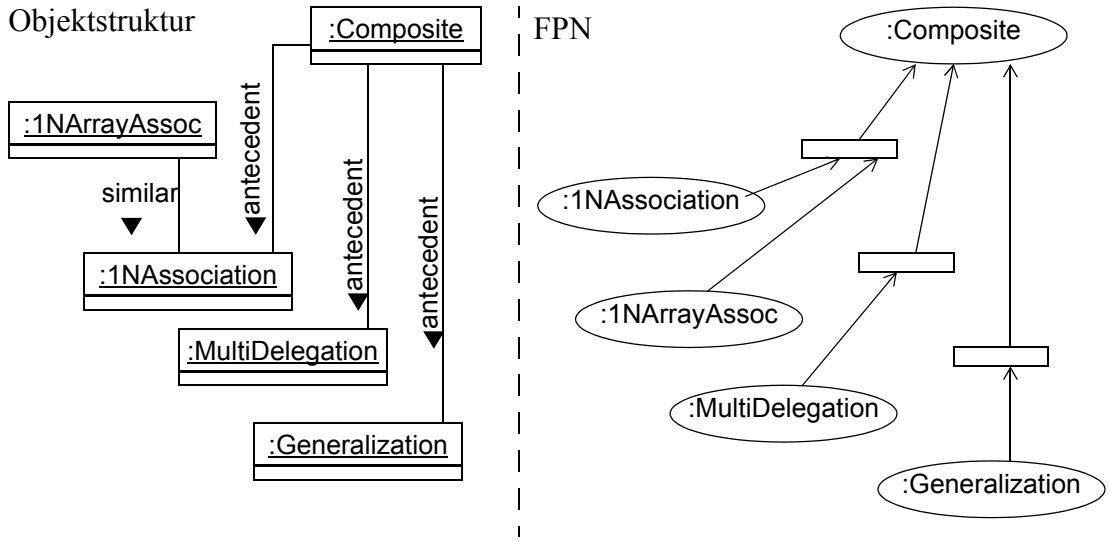


Abbildung 5.30 Beispiel für Abbildung Objektstruktur auf Fuzzy-Petrinetz

Stellen des Fuzzy-Petrinetzes erhalten die Namen der Annotationen, um die Korrespondenz zu verdeutlichen.

Die Kantenmenge *F* verbindet Stellen und Transitionen beziehungsweise Transitionen mit Stellen. Für einen antecedent-Link zwischen zwei Annotationen wird also eine Transition erzeugt und diese mit den entsprechenden Stellen für die Annotationen verbunden. Sollte ein similar-Link bestehen, so wird eine zusätzliche Kante von der Stelle zur Transition gezogen. Abbildung 5.30 zeigt an einem Teilausschnitt des Beispiels aus Abbildung 5.29 die Korrespondenz zwischen dem Ergebnis des Inferenzalgorithmus und dem erzeugten Fuzzy-Petrinetz. Für den similar-Link zwischen :1NArrayAssoc und :1NAssociation ist eine Kante von der Stelle :1NArrayAssoc zur Transition, die dem antecedent-Link zur :Composite-Annotation entspricht, gezogen worden.

Die Funktionen *bf* und *th* weisen jeder Stelle den Genauigkeits- und den Schwellwert der Musterinstanzregel zu, die die Annotation erzeugt hat. Die partielle Funktion *ubm* wird verwendet, um Änderungen, die der Reengineer an Präzisionswerten von Annotationen vorgenommen hat, auf die der Annotation zugehörigen Stelle im Fuzzy-Petrinetz zuzuweisen. Berechnete fuzzy-belief-markings werden durch die Funktion *cbm* jeder Stelle zugewiesen. Das eigentliche fuzzy-belief-marking wird durch die Funktion *fbm* einer Stelle zugewiesen. Sollte der Reengineer an einer Annotation einen Präzisionswert geändert haben, so wird dies durch die Funktion übernommen. Das berechnete fuzzy-belief-marking hat keinen Einfluss und wird quasi vom Reengineer überschrieben. Durch den Erhalt des berechneten fuzzy-belief-marking sind Vergleiche zwischen dem benutzerdefinierten Präzisionswert als Sollwert und dem berechneten Istwert möglich.

5.4.2 Ausführung des Fuzzy-Petrinetzes

Da für jede Annotation eine Stelle im FPN erzeugt worden ist, entsprechen ihre Präzisionswerte dem fuzzy-belief-marking, die einer Stelle des Netzes einen Wert [0..1] zuweist. Die Berechnung des fuzzy-belief-markings erfolgt gemäß [KM96] in zwei Phasen. In der ersten Phase wird für jede Transition ein Fuzzy-Truth-Token bestimmt.

Definition 5.5: Fuzzy-Truth-Token

Das Fuzzy-Truth-Token ist durch die Funktion *ftt:* $T \rightarrow [0..1]$ gegeben mit:

$\forall\ t \in T$ ist ftt (t) = Max ({fbm(s) | (s,t) $\in$ F})

Für jede Transition des Fuzzy-Petrinetzes ergibt sich das Fuzzy-Truth-Token als Maximum des fuzzy-belief-markings der Stellen im Vorbereich. Durch die oben beschriebene Konstruktion des Fuzzy-Petrinetzes hat eine Transition nur dann mehr als eine Stelle im Vorbereich, wenn ein similar-Link durch den Inferenzalgorithmus erzeugt worden ist. Wenn also alternative Annotationen existieren, so wird von diesen Alternativen der größte Präzisionswert gewählt. Im Beispiel in Abbildung 5.30 kann also entweder die Annotation a:1NAssociation oder a1:Association ihr fuzzy-belief-marking an die Transition weitergeben.

Definition 5.6: Berechnung des fuzzy-belief-markings

Die *cbm* Funktion ergibt sich für jede Stelle wie folgt: $\forall\ s \in S$ ist

- $cbm\ (s) = bf(s)$ *wenn* $\neg\exists\ t \in T$ *mit* $(t,s) \in F$
- $cbm\ (s) = 0$ *wenn* $Min\ (\{ftt(t) \mid (t,s) \in F\}) \leq th(s)$
- $cbm\ (s) = Min\ (\{bf(s), ftt(t) \mid (t,s) \in F\})$ *sonst*

In der zweiten Phase erfolgt die Berechnung des fuzzy-belief-markings der Stellen. Dabei werden drei Fälle unterschieden. Handelt es sich um eine Stelle, die keine Transition in ihrem Vorbereich besitzt, so entspricht das berechnete fuzzy-belief-marking dem Genauigkeitswert der zur Annotation der Stelle gehörenden Musterinstanzregel. Dieser Fall tritt für alle Annotationen, die von Rang-0 Regeln erzeugt worden sind, auf.

Im zweiten Fall existiert ein Fuzzy-Truth-Token im Vorbereich der Stelle, das einen kleineren Wert besitzt als der Schwellwert der entsprechenden Musterinstanzregel. Der Wert 0 sagt dabei aus, dass das Ergebnis nicht relevant ist und ignoriert werden kann.

Im dritten Fall ergibt sich das fuzzy-belief-marking der Stelle als Minimum des Genauigkeitswerts der zugehörigen Musterinstanzregel und aller Fuzzy-Truth-Token im Vorbereich der Stelle.

Die Berechnung des Präzisionswerts einer Annotation entspricht dabei der Berechnung des Präzisionswerts aus Definition 4.11.

Die Maximumbildung bei der Berechnung des Fuzzy-Truth-Token und die Minimumbildung bei der Berechnung des fuzzy-belief-markings entspricht der Berechnung der Fuzzywerte von Ableitungen in Fuzzygrammatiken [Zad78]. Damit entspricht die Berechnung des Präzisionswerts durch das Fuzzy-Petrinetz der Berechung des Präzisionswerts einer Annotation gemäß Definition 4.11. Allerdings werden im Gegensatz zur Semantikdefinition eines Regelkatalogs im vorherigen Kapitel Annotationen mit Präzisionswert 0.0 erzeugt. Dies hat den Nachteil, dass gegebenenfalls mehr Annotationen erzeugt werden. Der Vorteil ist, dass der Reengineer sich auch diese “Fehlschläge” ansehen und weitere Schlüsse ziehen kann.

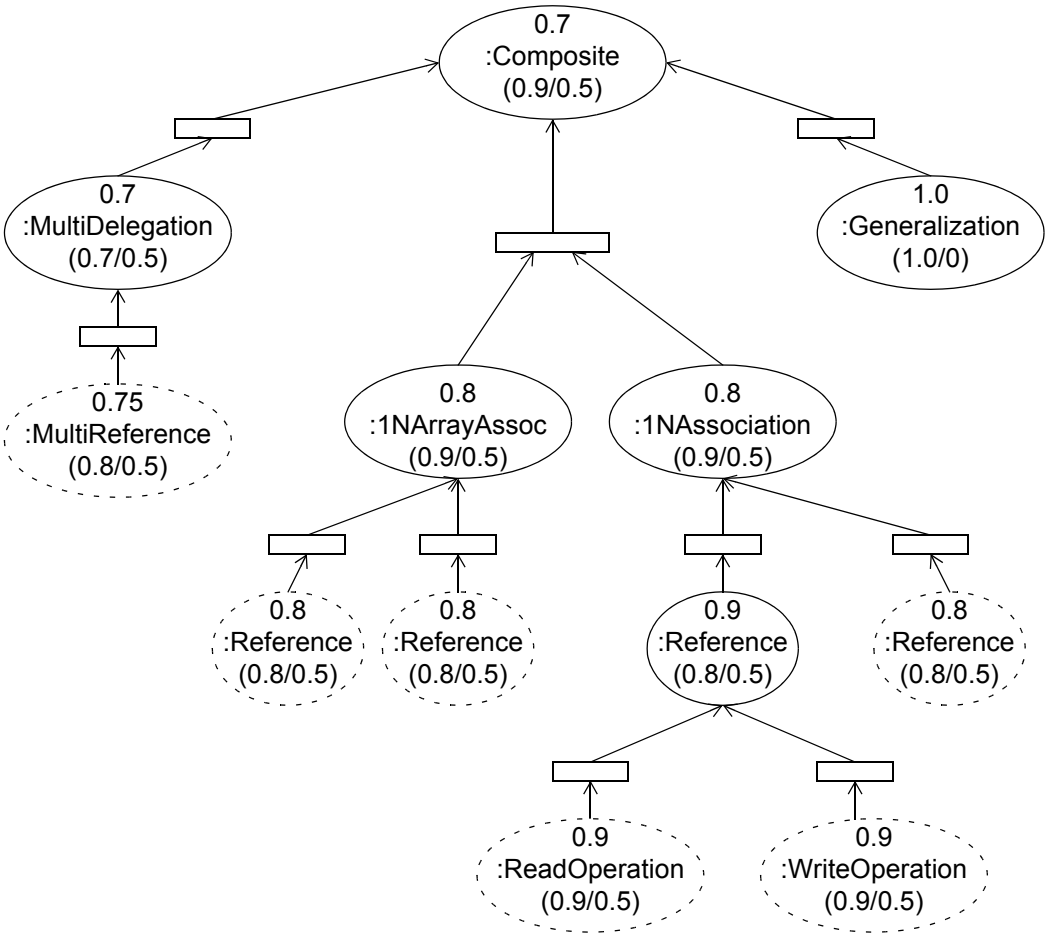


Abbildung 5.31 Fuzzy-Petrinetzbeispiel

Das konstruierte Fuzzy-Petrinetz enthält keine Zyklen, wodurch die Auswertung des Fuzzy-Petrinetzes durch eine einfache Vorwärtsevaluation der Stellen und Transitionen erfolgen kann. Ausgangspunkt für die Evaluation sind diejenigen Stellen, die keine Transitionen in ihrem Vorbereich besitzen. Dies sind die Stellen, die zu Annotationen gehören, die durch Rang-0 Regeln erzeugt worden sind.

Abbildung 5.31 zeigt das Fuzzy-Petrinetz für den Beispielausschnitt der durch den Inferenzalgorithmus erzeugten Objektstruktur aus Abbildung 5.29. Die Stellen des Fuzzy-Petrinetzes enthalten neben den Namen der korrespondierenden Stellen den berechneten Präzisionswert oberhalb des Namens. Unterhalb des Namens sind der Genauigkeitswert und der Schwellwert der entsprechenden Musterinstanzregel dargestellt. Die gestrichelt dargestellten Stellen sollen andeuten, dass das Netz an diesen Stellen nicht vollständig ist, sondern sich noch weitere Transitionen und Stellen anschließen. Es handelt sich dabei nur um eine graphische Darstellung.

Nach der Berechnung der Präzisionswerte werden diese in die Objektstruktur auf die Annotationen übertragen, siehe Abbildung 5.32. Alternativ zur Konstruktion des Fuzzy-Petrinetzes kann die Berechnung der Präzisionswerte der Annotationen direkt aufgrund der erzeugten Objektstruktur des Inferenzalgorithmus erfolgen. Aus technischer Sicht ist ein solches Vorgehen effizienter als die explizite Konstruktion des Fuzzy-Petrinetzes.

5.4.3 Änderungen des Reengineer

Nach der Berechnung der Präzisionswerte für alle erzeugten Annotationen hat der Reengineer die Möglichkeit, sich das Ergebnis anzusehen und gegebenenfalls Änderungen an Präzisionswerten einzelner Annotationen zu machen. Einzelne Änderungen von Präzisionswerten fließen in das Fuzzy-Petrinetz durch die Funktion *ubm* aus Definition 5.4 ein. Die Werte der *ubm*-Funktion werden direkt in die Berechnung übernommen. Dabei haben Änderungen an Präzisionswerten nur Einfluss auf Transitionen und Stellen im transitiven Nachbereich der geänderten Stelle.

Abbildung 5.32 zeigt die Beispielobjektstruktur aus Abbildung 5.29 mit den errechneten Präzisionswerten. Die Präzisionswerte werden im value-Attribut jedes Annotationsobjekts gespeichert. Änderungen des Reengineer sind durch Blitze angedeutet, und der neue Wert ist rechts daneben angegeben.

Die Korrektur des Präzisionswerts von 0.8 auf 0.9 der :1NArrayAssoc-Annotation hat keinen Einfluss auf Präzisionswerte der Annotationen im Nachbereich. Die Änderung bewirkt nur eine Erhöhung des Werts der nachfolgenden Transition (Abbildung 5.31) aufgrund der Maximumbildung. Auf den Präzisionswert der :Composite-Annotation hat diese Änderung keinen Einfluss, weil der Präzisionswert der :MultiDelegation-Annotation einen Wert von 0.7 hat und somit bei der Minimumbildung zur Berechnung des Präzisionswerts verwendet wird. Bei der zweiten Korrektur des Reengineer wird der Präzisionswert der :MultiDelegation-Annotation von 0.7 auf 1.0 angehoben. Durch diese Änderung erkennt der Reengineer die gefundene MultiDelegationinstanz als solche an. Da der Präzisionswert bei der Minimumberechnung für den Präzisionswert der :Composite-Annotation ausschlaggebend war, muss der Präzisionswert der :Composite-Annotation erneut berechnet werden. Der reevaluierte Präzisionswert - angedeutet durch den Strich - ist 0.9, der Präzisionswert der zuvor geänderten :1NArrayAssoc-Annotation.

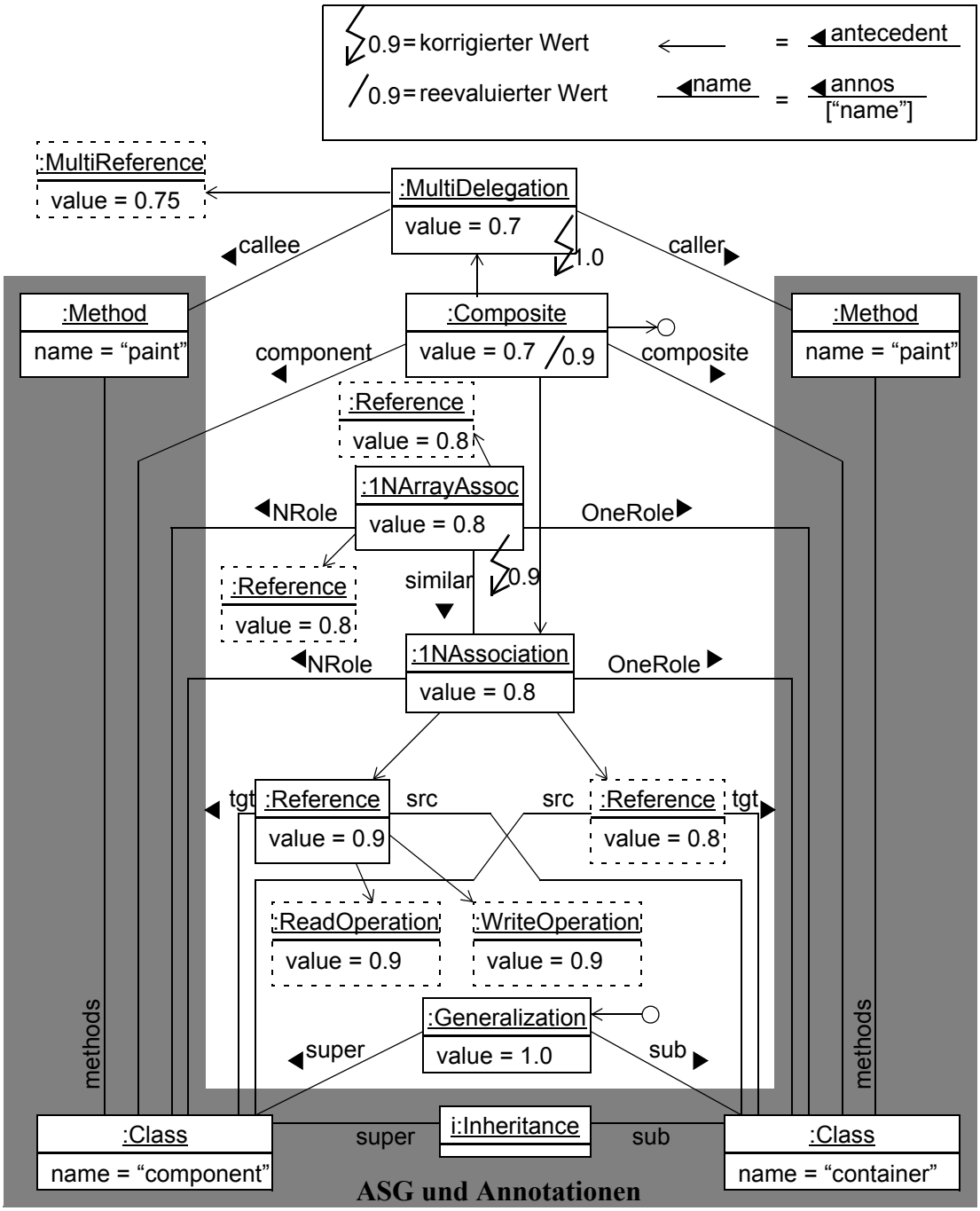


Abbildung 5.32 Objektstrukturbeispiel mit Präzisionswerten und Korrekturen

Allgemein gilt, dass eine Korrektur eines Präzisionswerts Auswirkungen auf Präzisionswerte von Annotationen im transitiven Nachbereich der geänderten Annotation hat.

Sonderfälle sind die Korrektur eines Präzisionswerts einer Annotation über oder unter den Schwellwert. Bei Änderung über den Schwellwert muss die Annotation in die bottom-up Queue einsortiert werden und eine Änderung unter den Schwellwert hat Auswirkungen auf Annotationen im Nachbereich. Durch eine solche Änderung können die Annotation im Nachbereich der geänderten Annotation den Präzisionswert 0.0 gemäß Definition 5.4 des Fuzzy-Petrinetzes erhalten. Der Präzisionswert 0.0 einer Annotation hat zur Folge, dass prinzipiell alle Annotationen im Nachbereich ebenfalls den Wert 0.0 erhalten. Eine Änderung des Präzisionswerts der :MultiReference-Annotation von 0.75 auf 0.45 hätte demnach zur Folge, dass die :MultiDelegation-Annotation und die :Composite-Annotation den Präzisionswert 0.0 erhielte. Eine Änderung des Präzisionswerts der :ReadOperation-Annotation unterhalb des Schwellwerts von 0.5 hätte demgegenüber nur Einfluss auf die Präzisionswerte der :Reference- und :1NAssociation-Annotation. Der Präzisionswert der :Composite-Annotation würde sich nicht ändern, weil zwischen der :1NAssociation-Annotation und der :1NArrayAssoc-Annotation ein similar-Link existiert, und in diesem Fall der Präzisionswert der :1NArrayAssoc-Annotation ausschlaggebend für den Präzisionswert der :Composite-Annotation ist.

5.5 Zusammenfassung

Die inkrementelle automatische Analyse, die Musterinstanzregeln eines Regelkatalogs auf eine Abstraktsyntaxgraphrepräsentation des Quelltextes anwendet, erfolgt in zwei Schritten. Im ersten Schritt werden Annotationen erzeugt, dessen Präzisionswerte im zweiten Schritt berechnet werden.

Der Inferenzalgorithmus verwendet eine Kombination aus bottom-up und top-down Verfahren, um möglichst schnell relevante Ergebnisse zu produzieren. Relevante Ergebnisse sind dabei Annotationen, die Entwurfsmusterinstanzen repräsentieren. Der Inferenzalgorithmus kann im bottom-up Modus unterbrochen und später fortgesetzt werden.

Die Reihenfolge der Regelanwendung wird dabei durch den Regeltriggergraphen, der aus einem Regelkatalog konstruiert wird, vorgegeben. Außerdem wird eine optimierte Regelanwendung verwendet, mit deren Hilfe die Teilgraphensuche im durchschnittlichen Fall auf polynomielle beziehungsweise lineare Laufzeit reduziert werden kann. Der Algorithmus selbst ist durch Story-Pattern beschrieben, dessen formale Definition auf [Zün01] beruht.

Die Berechnung der Präzisionswerte der erzeugten Annotationen erfolgt durch ein zyklenfreies Fuzzy-Petrinetz im Anschluss oder bei einer Unterbrechung des Inferenzalgorithmus. Änderungen, die der Reengineer an einzelnen Präzisionswerten vornimmt, werden in das Fuzzy-Petrinetz übernommen und überschreiben berechnete Werte. Wie Korrekturen des Reengineer an Präzisionswerten der Ergebnisse in die Genauigkeits- und Schwellwerte der Musterinstanzregeln einfließen, wird im folgenden Kapitel vorgestellt.

KAPITEL 6: AUTOMATISCHE ADAPTION

Ein Problem des hier vorgestellten semi-automatischen Reverse-Engineering Ansatzes ist die Wahl des Genauigkeits- und des Schwellwerts einer Musterinstanzregel. Beide Werte können vom Reengineer für jede Musterinstanzregel nur geschätzt werden. Insbesondere für die genaue Angabe des Genauigkeitswerts müsste der Reengineer Kenntnisse über das zu analysierende System haben, die er allerdings erst während des Analyseprozesses erhält.

Aufgrund des iterativen Analyseprozesses ist der Reengineer prinzipiell in der Lage, nach jeder Iteration den Genauigkeitswert und auch den Schwellwert einer Musterinstanzregel zu verändern. Durch die Trennung des Inferenzalgorithmus und Berechnung der Präzisionswerte für Annotationen kann eine erneute Berechnung mit den neuen Genauigkeits- und Schwellwerten stattfinden, ohne den Inferenzalgorithmus erneut zu starten. Der Reengineer kann dadurch die Genauigkeits- und Schwellwerte der Musterinstanzregeln und damit die resultierenden Präzisionswerte der Annotationen verändern. Die Korrektur der Genauigkeits- und Schwellwerte erfolgt dabei manuell.

Im Bereich der Wissensbasierten Systeme werden künstliche Adaptions- beziehungsweise Lernsysteme verwendet, um die Adaption der Werte zu automatisieren. In diesem Kapitel wird ein erster Ansatz zur Adaption der Genauigkeits- und Schwellwerte von Musterinstanzregeln vorgestellt. Der Ansatz basiert auf einfachen statistischen Berechnungen durch Soll-Istwert Vergleiche. Im folgenden Abschnitt werden Grundlagen von Adaptions- beziehungsweise Lernsystemen beschrieben, bevor anschließend die Adaption des Genauigkeitswerts und des Schwellwerts einer Musterinstanzregel vorgestellt wird.

6.1 Lernen durch Beispiele

Künstliche Lern- beziehungsweise Adaptionssysteme, wie sie in technischen Systemen verwendet werden, basieren auf statistischen Schätzungen. Allgemein wird auf Grundlage von Trainingsdaten versucht, Aussagen über zukünftige Testdaten zu machen. Dazu müssen Trainings- sowie Testdaten auf der gleichen statistischen Basis beruhen. Das heißt, dass die statistische Basis sich nicht ändern darf. Cherkassy und Mulier [CM98] unterscheiden drei Aufgabenbereiche:

- Klassifikation beziehungsweise Schätzung der Klassifikationsgrenzen
- Regression beziehungsweise Schätzung einer Funktion auf Basis von Beispielen
- Schätzung der Wahrscheinlichkeitsdichten

Zusätzlich existieren zwei verschiedene Arten des Lernens. Es wird zwischen *nicht überwachtem Lernen* (englisch: unsupervised learning) und *überwachtem Lernen* (englisch: supervised learning) unterschieden. Beim nicht überwachten Lernen wird dem System lediglich eine Menge von Eingaben zur Verfügung gestellt. Ausgaben werden nicht betrachtet. Nicht überwachtes Lernen wird häufig zur Schätzung von Wahrscheinlichkeitsdichten verwendet.

Beim überwachten Lernen werden Ein- und Ausgaben des Systems miteinander verglichen. Durch den Vergleich beziehungsweise eine Bewertung des Vergleichs lernt das System. Der Vergleich beziehungsweise die Bewertung kann dabei manuell oder automatisch durch die Vorgabe von Sollwerten erfolgen.

Bei dem in diesem Kapitel vorgestellten Adaptionssystem handelt es sich um überwachtes Lernen. Die Bewertung erfolgt manuell durch den Reengineer, der Präzisionswerte einzelner Ergebnisse - Annotationen - ändern kann. Die Adaption der Genauigkeits- und Schwellwerte der Musterinstanzregeln erfolgt durch *Regression* und wird allgemein durch die in Abbildung 6.1 dargestellte Formel repräsentiert.

$$y = f(x, w) + e$$

Abbildung 6.1 Formel für Regression

Dabei bezeichnet x den Eingangsvektor, y den Ausgangsvektor, w den Parametervektor und e den Vektor der Fehlerterme. Die Funktion f wird als Approximationsfunktion bezeichnet und ist oft keine geschlossen formulierbare Funktion. Ziel der Regression ist es, den Fehlerterm e durch die Adaption des Parametervektors w zu minimieren. Dabei wird w aufgrund statistischer Daten geschätzt.

Übertragen auf den hier vorgestellten Ansatz wird die Funktion f durch die Objektstrukturen mit den erzeugten Annotationen und Beziehungen während des Inferenzalgorithmus beschrieben. Der Parametervektor w besteht aus den Genauigkeits- und Schwellwerten der Musterinstanzregeln. Eingabe x ist der zu analysierende Quelltext, und die Ausgabe y sind die Präzisionswerte der Ergebnisse. Der Fehlerterm ist durch die Korrekturen des Reengineer an den Präzisionswerten gegeben. Das Ziel der Regression ist es also, die Genauigkeits- und Schwellwerte der Musterinstanzregeln so anzupassen, dass der Reengineer keine beziehungsweise lediglich kleine Änderungen an den Präzisionswerten der Annotationen vornehmen muss, die durch den Inferenzalgorithmus erzeugt worden sind. Dies entspricht der Minimierung des Fehlervektors e.

Im Bereich der künstlichen Lern- beziehungsweise Adaptionssysteme werden allgemein numerische Verfahren verwendet. Äquivalente Darstellungen solcher numerischer Verfahren sind Neuronale-Netze. Neuronale-Netze werden zum Beispiel zur Handschriftenerkennung, Bilderkennung angewendet. Allgemein existieren eine Reihe von

Veröffentlichungen zur Verwendung Neuronaler-Netze zur allgemeinen Mustererkennung (englisch: pattern-recognition) [Pao89, Kos92, BH94, Bis95, Rip96].

Im VARLET-Projekt, siehe Kapitel 2.4.5, das sich mit dem Reengineering von Datenbanken beschäftigt, ist ein neuronales Netz verwendet worden, um Fuzzywerte zu adaptieren [JS99]. Es werden Generic Fuzzy Reasoning Nets (GFRN) zur Analyse von Datenbanken verwendet. Das GFRN beruht auf possibilistischer Logik [Zad78, DLP94], und die Berechnung der Fuzzywerte erfolgt wie in diesem Ansatz durch ein Fuzzy-Petrinetz.

Zur Adaption der Fuzzywerte des GFRN werden Fuzzy-Neuronale-Netze verwendet [Str99]. Das Lernen der Adaptionswerte erfolgt durch stochastische Approximation, dem so genannten *Gradienten Verfahren* (englisch: gradient descent), siehe [Wer94].

Die Lernaufgaben - Trainingsdaten - werden aus korrigierten Analyseergebnissen ermittelt. Aus einem Analyseergebnis können dabei eine Menge von Lernaufgaben entstehen. Eine Lernaufgabe ist dabei vergleichbar mit der Rückverfolgung eines Ableitungswegs einer Annotation bis zu einem Objekt des abstrakten Syntaxgraphen. Die Menge aller Lernaufgaben eines Ergebnisses besteht dann aus allen Wegen von Annotationen zu Objekten des abstrakten Syntaxgraphen. Bei der Analyse großer Systeme explodiert dabei die Anzahl der Lernaufgaben, und die Adaption hat eine hohe Laufzeit. Die hohe Laufzeit der Adaption ist ein Ergebnis aus [Str99], das bereits bei kleinen Datenbanksystemen zu beobachten ist.

Ein Problem, dass allgemein bei der Verwendung Neuronaler-Netze besteht, ist die Veränderung des Netzes. Allgemein wird bei künstlichen Lern- beziehungsweise Adaptionssystemen davon ausgegangen, dass sich die Struktur der Netze nicht stark ändert. Bei [JS99] ändert sich das Fuzzy-Neuronale-Netz nicht, weil sich das verwendete GFRN während der Analyse nicht ändert.

In dem hier vorgestellten Ansatz zur Erkennung von Entwurfsmusterinstanzen in Quelltexten großer Systeme ist allerdings die Idee des Ansatzes, dass sich der Regelkatalog während des Analyseprozesses ändert. Außerdem können Regelkataloge, die für ein bestimmtes Softwaresystem angepasst sind, nicht einfach für andere Systeme verwendet werden. Dadurch können bereits analysierte Systeme nicht unbedingt als Trainingsdaten verwendet werden.

Aufgrund der teilweise langen Laufzeit einer Adaption und der Änderungen des Regelkatalogs wird zur Adaption der Genauigkeits- und Schwellwerte der Musterinstanzregeln eines Regelkatalogs ein einfaches Vergleichsverfahren verwendet.

6.2 Adaption durch statistisches Verfahren

Schwellwert und Genauigkeitswert einer Musterinstanzregel hängen eng zusammen, was eine Verknüpfung der Adaption beider Werte auf Basis einer Korrektur des Reengineer nahelegt. Eine Verknüpfung der Adaption erfordert dabei eine Untersuchung des Korrekturverhaltens des Reengineer, siehe [Nie01, Nie02, NWW03]. Grundlage für eine solche

Untersuchung kann zum Beispiel die hier verwendete getrennte Adaption von Genauigkeits- und Schwellwert sein.

Die Adaption des Genauigkeits- und Schwellwerts einer Musterinstanzregel erfolgt durch ein statistisches Vergleichsverfahren. Dabei werden Korrekturen des Reengineer als Sollwert und die berechneten Werte als Istwerte betrachtet. Der Reengineer kann die Ergebniswerte der automatischen Analyse dabei auf zwei unterschiedliche Arten korrigieren. Dabei wird die erste Korrekturart für die Adaption der Genauigkeitswerte und die zweite Korrekturart für die Adaption der Schwellwerte verwendet.

6.2.1 Korrektur der Präzisionswerte

Der Istwert für den Vergleich zur Adaption des Genauigkeitswerts einer Musterinstanzregel ist der Präzisionswert einer Annotation, der durch das Fuzzy-Petrinetz, siehe Kapitel 5.4, berechnet worden ist. Der Sollwert für den Vergleich wird vom Reengineer durch die Korrektur eines Präzisionswerts einer Annotation vorgenommen.

Die Korrektur eines Präzisionswerts durch den Reengineer hat nicht nur Einfluss auf die Adaption des Genauigkeitswerts der zugehörigen Musterinstanzregel selbst, sondern beeinflusst auch den Genauigkeitswert der Musterinstanzregeln, die zu Annotationen im direkten Vorbereich[1] der korrigierten Annotation gehören. Genauigkeitswerte von Musterinstanzregeln von Annotationen im Nachbereich der korrigierten Annotation nehmen auf die Korrektur keinen Einfluss. Bei der Korrektur der Präzisionswerte von Annotationen werden zwei Fälle unterschieden - die Erhöhung beziehungsweise die Verringerung eines Präzisionswerts einer Annotation. Beide Änderungen haben Einfluss auf den Genauigkeitswert der Musterinstanzregeln von Annotationen im Vorbereich. Der Einfluss der Korrektur eines Präzisionswerts auf Annotationen im Vorbereich erfolgt dabei auf Grundlage der Minimumbildung bei der Berechnung des Präzisionswerts der Annotation. Die Erhöhung beziehungsweise die Verringerung eines Präzisionswerts muss dabei getrennt betrachtet werden.

Erhöhung eines Präzisionswerts

Abbildung 6.2 zeigt einen Ausschnitt des Fuzzy-Petrinetzes, an dem die Propagation der Korrektur verdeutlicht werden soll. Zur besseren Verständlichkeit besitzen die Stellen des Fuzzy-Petrinetzes sowohl den Namen der entsprechenden Annotation, als auch oberhalb den aktuell berechneten Präzisionswert. Die dargestellte Situation entspricht einem Ausschnitt des Fuzzy-Petrinetzes aus Abbildung 5.31.

Im Beispiel korrigiert der Reengineer den Präzisionswert der :Composite-Stelle von 0.7 auf 0.85. Der in Klammern stehende Wert ist die relative Veränderung des aktuellen Präzisionswerts zum korrigierten Wert. Der berechnete Präzisionswert 0.7 ist durch die Bil-

1. Der Vorbereich einer Annotation a sind alle Annotationen, die für die Anwendung der Musterinstanzregel zur Erzeugung der Annotation a verwendet worden sind.

dung des Minimums über alle Präzisionswerte der Annotationen im Vorbereich entstanden, in diesem Fall durch die :MultiDelegation-Stelle. Durch die Erhöhung des Präzisionswerts der :Composite-Annotation wird die relative Erhöhung an allen Präzisionswerten der Annotationen im Vorbereich, die kleinere Werte als den korrigierten besitzen, propagiert. Im Beispiel sind dies die :1NArrayAssoc-, die :1NAssociation- und die :MultiDelegation-Stelle. Durch die Propagation der relativen Erhöhung wird bei einer Minimumbildung an der korrigierten Stelle der gewünschte Wert "berechnet".

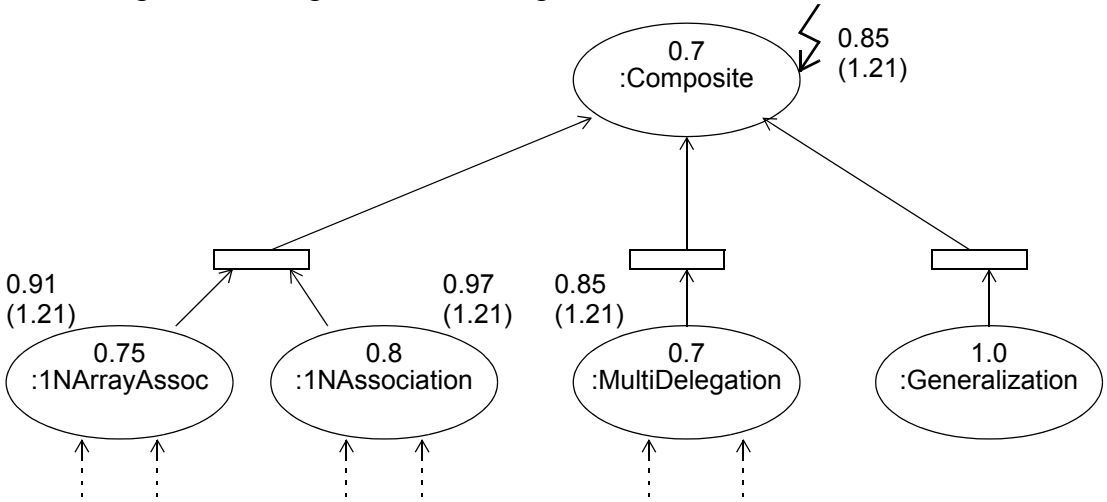


Abbildung 6.2 Beispiel für eine Erhöhung eines Präzisionswerts

Die Korrektur an der :Composite-Annotation erfolgt dabei sofort, während die durch Propagation entstandenen Präzisionswerte für Annotationen von Vorgängerregeln gesammelt werden; so genannte *virtuelle Präzisionswerte*. Durch die Propagation der relativen Erhöhung können dabei virtuelle Präzisionswerte entstehen, die größer sind als 1.0 und damit den Wertebereich verlassen. Dies muss bei der Adaption, bei der alle virtuellen Präzisionswerte verrechnet werden, berücksichtigt werden. Durch die virtuellen Präzisionswerte wird außerdem eine rekursive Propagation von Korrekturwerten verhindert, die gegebenenfalls andere Korrekturen, die bereits vorgenommen worden sind, überschreibt.

Die Propagation der relativen Werte ist dabei nur eine Möglichkeit, die bei der Adaption der Genauigkeitswerte verwendet werden kann. Zum Beispiel könnten absolute Werte propagiert werden, oder höhere Werte könnten nach unten korrigiert werden. Das hier verwendete Propagationsverfahren hat in der Praxis allerdings gute Ergebnisse geliefert.

Verringerung eines Präzisionswerts

Abbildung 6.3 zeigt die gleiche Situation wie vorher, nur dass der Reengineer in diesem Fall den Präzisionswert der Annotation :Composite von 0.7 auf 0.6 verringert. Die relative Verringerung wird dabei an alle Annotationen im Vorbereich propagiert, dessen berechneter Präzisionswert größer als der korrigierte Wert ist. In diesem Fall erhalten die

Annotationen :1NAssociation, :MultiDelegation und :Generalization einen virtuellen Präzisionswert, wohingegen die Annotation :1NArrayAssoc keinen virtuellen Präzisionswert erhält. Aufgrund der Propagation der relativen Verringerung würde - wie bei der Erhöhung eines Präzisionswerts - bei einer Minimumbildung an der korrigierten Stelle der gewünschte Wert “berechnet”.

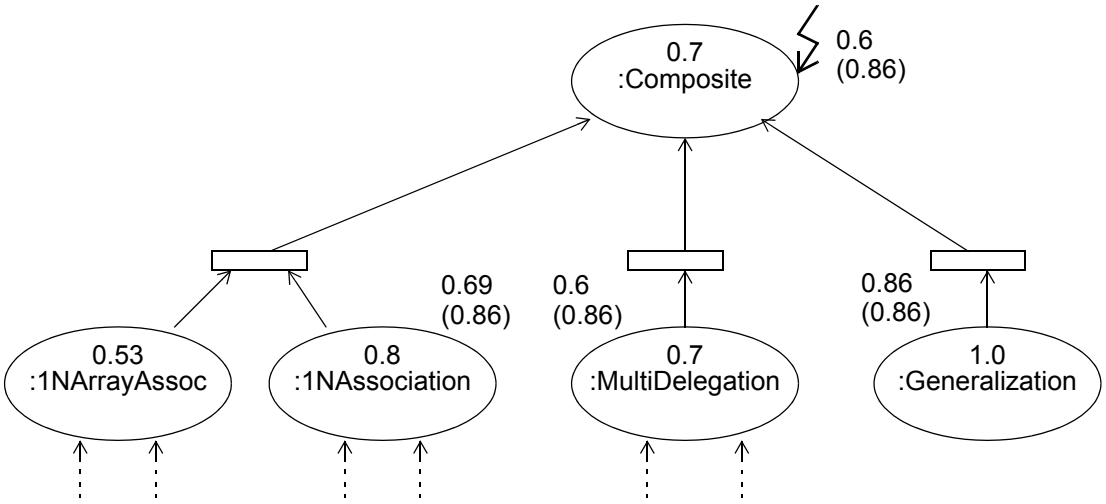


Abbildung 6.3 Beispiel für eine Verringerung eines Präzisionswerts

Durch die Propagation der relativen Verringerung entstehen keine virtuellen Präzisionswerte mit einem Wert kleiner als 0.0. Der Wertebereich wird also nicht verlassen. Allerdings kann die Situation entstehen, dass durch die Korrektur ein Präzisionswert kleiner ist als der Schwellwert der zur Annotation gehörenden Musterinstanzregel. In diesem Fall findet keine Propagation der relativen Verringerung statt.

Sowohl bei der Erhöhung als auch bei der Verringerung eines Präzisionswerts werden gegebenenfalls relative Werte an Annotationen im Vorbereich propagiert. Jede Propagation erzeugt dabei einen virtuellen Präzisionswert für eine Annotation. Solange der Reengineer Korrekturen vornimmt, entsteht also für eine Annotation eine Menge virtueller Präzisionswerte.

6.2.2 Korrektur der Schwellwerte

Der Schwellwert einer Musterinstanzregel dient dazu, unpräzise Ergebnisse - Annotationen der Musterinstanzregel - mit niedrigem Präzisionswert, zu filtern. Der Reengineer hat daher nur die Möglichkeit, Ergebnisse zu bestätigen oder zu revidieren und somit Einfluss auf die automatische Adaption des Schwellwerts einer Musterinstanzregel zu nehmen.

Abbildung 6.4 zeigt zwei Beispiele für Korrekturen. Im ersten Fall liegt der berechnete Präzisionswert [0.45] unterhalb des Schwellwerts, der als Genauigkeitswert/Schwellwert Paar 0.9/0.5 unterhalb des Annotationsnamens dargestellt ist. Der Genauigkeitswert ist daher in eckigen Klammern dargestellt. Der Reengineer kann das Ergebnis dahingehend

korrigieren, dass er die Korrektheit des Ergebnisses bestätigt, ohne allerdings den Präzisionswert der Annotation zu ändern. Dies hat einen virtuellen Schwellwert von 0.45 für die Annotation zur Folge. Außerdem wird die Annotation zur Anzeige gebracht, auch wenn der Schwellwert der Musterinstanzregel dies nicht erlaubt. Auf die Berechnung der Präzisionswerte und Schwellwerte anderer Annotationen hat die Korrektur keinen Einfluss. Ebenso werden keine relativen Werte propagiert. Es handelt sich also lediglich um eine auf die Annotation beschränkte Korrektur.

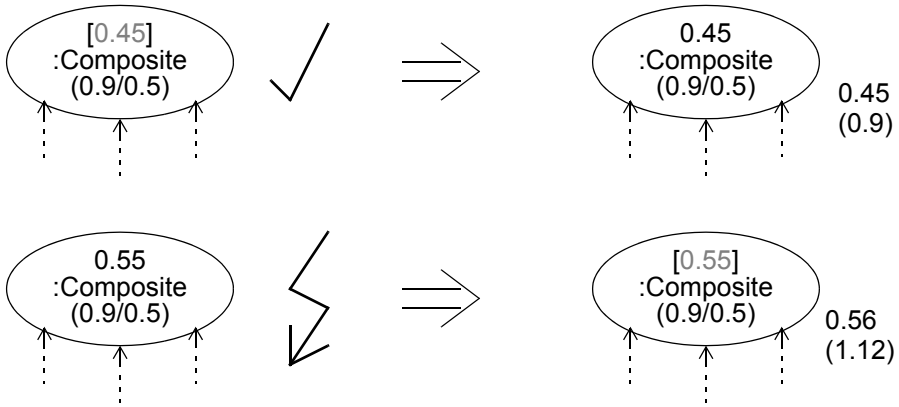


Abbildung 6.4 Schwellwertkorrekturbeispiel

Im zweiten Beispiel in Abbildung 6.4 revidiert der Reengineer das Ergebnis, was einen virtuellen Schwellwert von 0.56 zur Folge hat. Außerdem wird das Ergebnis aus der Anzeige entfernt - angedeutet durch die Klammern. Ebenso wie im ersten Fall hat die Korrektur keinen Einfluss auf Berechnungen anderer Präzisionswerte oder Schwellwerte.

Prinzipiell hätte der Reengineer, anstatt das Ergebnis zu bestätigen beziehungsweise zu korrigieren, den Präzisionswert der Annotation entsprechend erhöhen beziehungsweise verringern können. Dies hätte allerdings sowohl Auswirkungen auf die Präzisionswerte der Annotationen im Nachbereich bei einer Reevaluation des Fuzzy-Petrinetzes gehabt, als auch Einfluss auf die Adaption der Genauigkeitswerte der Musterinstanzregeln in Annotationen im Vorbereich genommen. Außerdem ist es dem Reengineer überlassen, sowohl den Präzisionswert zu ändern als auch das Ergebnis zu bestätigen oder zu revidieren. Beides erzeugt eine Menge virtueller Präzisionswerte beziehungsweise virtueller Schwellwerte, die bei der Adaption verrechnet werden.

6.2.3 Adaption von Genauigkeitswert und Schwellwert

Durch die Korrekturen des Reengineer sind an den Annotationen eine Menge virtueller Präzisionswerte und virtueller Schwellwerte entstanden. Diese virtuellen Werte sind die Basis für die Adaption des Genauigkeitswerts und Schwellwerts einer Musterinstanzregel. Im Folgenden wird die Berechnung des neuen Genauigkeitswerts einer Musterinstanzregel vorgestellt. Die Berechnung des neuen Schwellwerts der Regel ist identisch.

Die Adaption des Genauigkeitswerts einer Musterinstanzregel erfolgt in zwei Schritten. Zuerst wird der Mittelwert $\overline{v_{anno}}$ der virtuellen Präzisionswerte an jeder Annotation gebildet. Die virtuellen Präzisionswerte werden dabei in einer Liste $virValue_{anno}$ gespeichert. Für Annotationen, an denen keine virtuellen Präzisionswerte vorhanden sind, ist der Mittelwert gleich dem Präzisionswert der Annotation $value_{anno}$ selbst. Die Berechnung ist in Abbildung 6.5 dargestellt.

$$\overline{v_{anno}} = \begin{cases} \frac{1}{n}\sum_{i=1}^{n} virValue_{anno}[i] & \text{falls virtuelle Werte existieren} \\ value_{anno} & \text{sonst} \end{cases}$$

Abbildung 6.5 Berechnung virtueller Präzisionsmittelwerte

Die Mittelwertbildung im ersten Schritt der Berechnung ist nicht gewichtet, so dass alle Änderungen des Reengineer die gleiche Gewichtung haben. Auf diese Weise können sich Änderungen gegenseitig aufheben, obwohl sie in einem unterschiedlichen Kontext vorgenommen worden sind. Zum Beispiel könnte die Erhöhung des Präzisionswerts an einer Strategy-Annotation und gleichzeitig eine Verringerung des Präzisionswerts an einer Bridge-Annotation dazu führen, dass sich diese Änderungen an der gemeinsamen Reference-Annotation im Vorbereich aufheben.

Im ersten Schritt sind die Korrekturen an einzelnen Annotationen durch Mittelwertbildung verrechnet worden. Für die Adaption eines Genauigkeitswerts einer Musterinstanzregel werden im zweiten Schritt Korrekturen an Annotationen, die von der Musterinstanzregel erzeugt worden sind, miteinander verrechnet. Der neue Genauigkeitswert einer Musterinstanzregel bf_{rule} ist der Mittelwert über die *virtuellen Präzisionsmittelwerte* aller Annotationen, die durch die Musterinstanzregel erzeugt worden sind, siehe Abbildung 6.6. Mit $annos_{rule}$ wird dabei die Menge aller erzeugten Annotationen der Musterinstanzregel bezeichnet.

$$bf_{rule} = \frac{1}{|annos_{rule}|}\sum_{annos_{rule}} \overline{v_{anno}}$$

Abbildung 6.6 Berechnung des Genauigkeitswerts (Variante 1)

Der Vorteil dieser Berechnung ist, dass der berechnete Mittelwert abhängig ist von der Anzahl aller erzeugten Annotationen einer Musterinstanzregel. Je größer die Anzahl der Annotationen einer Musterinstanzregel bei gleicher Anzahl von Korrekturen, desto weniger Einfluss haben die Korrekturen auf die Adaption. Der Nachteil ist, dass der Reengineer dadurch prinzipiell alle Annotationen einer Musterinstanzregel bewerten muss, da

sonst die Adaption verfälscht sein könnte. Dieser Nachteil kann durch die etwas abgewandelte Variante in Abbildung 6.7 ersetzt werden.

$$bf_{rule} = \frac{1}{|inannos_{rule}|} \sum_{inannos_{rule}} \overline{v_{anno}}$$

Abbildung 6.7 Berechnung des Genauigkeitswerts (Variante 2)

In der 2. Variante wird der neue Genauigkeitswert einer Musterinstanzregel durch die Mittelwertbildung über alle Annotationen gebildet, die der Reengineer manuell inspiziert $inannos_{rule}$ hat. Inspizierte Annotationen sind Annotationen, an denen der Reengineer den Präzisionswert korrigiert hat oder an denen er den Präzisionswert der Annotation bestätigt. Sollte der Reengineer alle Annotationen einer Musterinstanzregel inspiziert haben, so ist die 2. Variante identisch mit der 1. Variante. Eine Ausnahme sind Annotationen von Musterinstanzregeln, an denen der Reengineer den Präzisionswert weder korrigiert noch bestätigt hat, allerdings virtuelle Präzisionswerte existieren. In diesem Fall gelten alle Annotationen dieser Musterinstanzregel als inspiziert. Durch diese Ausnahme wird verhindert, dass nur propagierte virtuelle Präzisionswerte für die Adaption der entsprechenden Musterinstanzregel verwendet werden und damit die Adaption verfälschen.

6.3 Zusammenfassung

Das vorgestellte Regressionsverfahren zur Adaption von Genauigkeits- und Schwellwerten der Musterinstanzregeln basiert auf einem Soll- Istwert Vergleich der korrigierten und berechneten Präzisionswerte der Annotationen. Korrekturen des Reengineer werden dabei auf Vorgängerannotationen propagiert. Die Adaption des Genauigkeits- und Schwellwerts einer Musterinstanzregel erfolgt durch eine Berechnung des Durchschnitts der Präzisionswerte aller von der Musterinstanzregel produzierten Annotationen.

Die Propagation von Präzisionswertänderungen an Vorgängerannotationen sowie die Berechnung der neuen Genauigkeits- und Schwellwerte kann aufgrund des verwendeten statistischen Verfahrens direkt nach der Änderung erfolgen. Die neuen Werte werden allerdings nur auf Verlangen des Reengineer in die Musterinstanzregeln übernommen, siehe Analyseprozess in Kapitel 3.2.

Erste Testergebnisse, die mit der hier vorgestellten automatischen Adaption der Genauigkeits- und Schwellwerte, durchgeführt worden sind, werden im nächsten Kapitel vorgestellt. Außerdem werden Anforderungen an eine technische Realisierung des in dieser Arbeit vorgestellten Ansatzes zur Erkennung von Entwurfsmusterinstanzen in Quelltexten großer Softwaresysteme beschrieben und praktische Erfahrungen mit einem Prototypen vorgestellt.

KAPITEL 7: PRAKTISCHE ERFAHRUNGEN

Der in den vorherigen Kapiteln beschriebene Ansatz zur Erkennung von Entwurfsmusterinstanzen in Quelltexten großer Systeme ist prototypisch implementiert worden. Zusätzlich zum Prototypen werden allgemeine Anforderungen, die für eine technische Realisierung des Ansatzes notwendig sind, vorgestellt. Es werden erste Ergebnisse von Skalierungstests der automatischen Analyse präsentiert. Außerdem wird ein Beispielvorgehen für ein neu zu analysierendes System präsentiert. Abschließend werden erste Ergebnisse und Auswirkungen der automatischen Adaption diskutiert.

7.1 Technische Realisierung

Eine technische Realisierung des in den vorherigen Kapiteln vorgestellten Ansatzes zur Erkennung von Entwurfsmusterinstanzen in Quelltexten großer Softwaresysteme erfordert unter anderem:

- einen Editor für Musterinstanzregeln sowie, die Möglichkeit, die Konsistenzbedingungen für einen Regelkatalog zu prüfen.
- einen Parser, der den Quelltext in einen abstrakten Syntaxbaum überführt.
- eine Implementierung des Inferenzalgorithmus und der Präzisionswertberechnung.
- eine Möglichkeit die Analyseergebnisse anzuzeigen.

Zur Darstellung der Musterinstanzregeln können zum Beispiel UML-Kollaborationsdiagramme verwendet werden, wobei den Objekten Stereotypen zugewiesen werden. Ebenso können für die Darstellung der Analyseergebnisse UML-Klassendiagramme verwendet werden. Annotationen können als solche in das Klassendiagramm eingetragen werden. Für die Eingabe der Musterinstanzregeln und die Anzeige der Ergebnisse kann ein UML-Entwicklungswerkzeug mit geeigneten Schnittstellen zum Datenaustausch wie zum Beispiel TOGETHERCONTROLCENTER [Tog], RATIONALROSE [Ros] oder FUJABA [FNT98] verwendet werden.

Für die Entwicklung des FINITE[1] Prototypen ist die FUJABA Entwicklungsumgebung verwendet worden. Der FINITE Prototyp ist dabei nicht nur mit FUJABA entwickelt worden, sondern erweitert die Entwicklungsumgebung um Reverse-Engineering Funktionalität [KNNZ99b, NNWZ00]. Die Entwicklungsumgebung FUJABA hat gegenüber anderen Entwicklungsumgebungen Vorteile. Erstens besitzt die Entwicklungsumgebung einen

1. FINITE ist ein von der Deutschen Forschungsgemeinschaft (DFG) gefördertes Projekt und ist die Abkürzung für Fuzzy Logic Basierte Interaktive Erkennung von Entwurfsmusterinstanzen.

Mechanismus, der es erlaubt, auf einfache Art und Weise die Entwicklungsumgebung zu erweitern und die bereits vorhandenen UML-Diagrammarten zu nutzen. Zweitens kann auf bereits vorhandene Graphikbibliotheken, Konsistenzmechanismen, etc. zurückgegriffen werden [ES89]. Drittens ist sie eine Implementierung des formalen Regelsystems aus [Zün01]. Damit kann der im vorherigen Kapitel vorgestellte Inferenzalgorithmus quasi direkt übernommen werden. Das Fuzzy-Petrinetz zur Berechnung der Präzisionswerte kann ebenfalls auf Basis des formalen Regelwerks beschrieben werden [Wen01] und kann damit ebenfalls direkt übernommen werden.

Regelkatalogeditor

Der Regelkatalogeditor ist eine Erweiterung von FUJABA und verwendet den zur Verfügung gestellten Plug-In Mechanismus. Regelkataloge und Musterinstanzregeln werden intern im Editor durch einen abstrakten Syntaxgraphen repräsentiert, siehe [Pal01]. Die Transformation in eine graphische Repräsentation erfolgt durch ein Model-View-Controler Konzept. Abbildung 7.1 zeigt den Musterinstanzeditor mit der Composite-Regel.

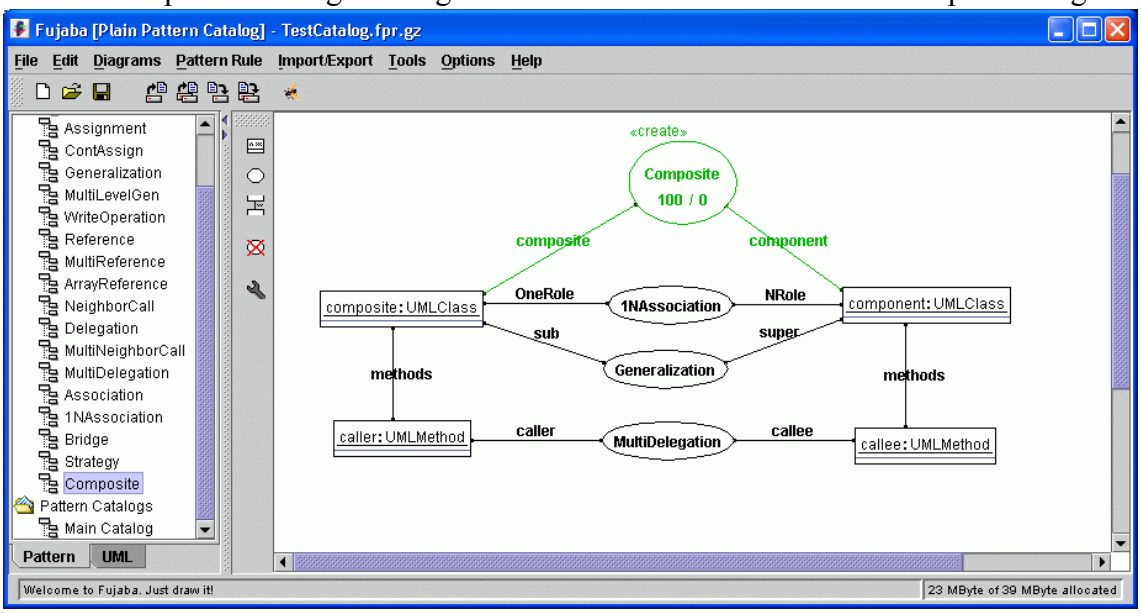


Abbildung 7.1 Regelkatalogeditor mit Composite-Regel

Abbildung 7.2 zeigt ein erweitertes Regeldefinitionsdiagramm des Regelkatalogs, wobei die Erweiterung aus der Darstellung der Abhängigkeitsbeziehungen zwischen den Musterinstanzregeln besteht. Das Regeldefinitionsdiagramm ist als UML-Klassendiagramm dargestellt, wobei Klassen, die Musterinstanzregeln repräsentieren, mit dem Stereotyp «PatternRule» gekennzeichnet sind. Die Hinzunahme der Regelabhängigkeiten in das Diagramm hat sich aus dem praktischen Einsatz des Prototypen ergeben. Zusammen mit einem Diagramm, dass die Annotationsbeziehungen der Regeln darstellt, hat der Reengineer einen besseren Überblick über die Musterinstanzregeln im Katalog. Die beiden Dia-

gramme unterstützen den Reengineer bei der Beurteilung und Überprüfung der Analyseergebnisse.

In der Praxis hat sich außerdem herausgestellt, dass ein Regelkatalogeditor nicht nur Funktionen zum Im- und Export und Ausschneiden-Kopieren-Einfügen von Musterinstanzregeln zur Verfügung stellen sollte, sondern diese Funktionen sich auch auf ganze Regelkataloge beziehen sollten. Insbesondere der Export von Teilregelkatalogen hat sich als notwendig erwiesen. Teilregelkataloge sind konsistent und können damit für die automatische Analyse verwendet werden. Sie sind in der Praxis häufig verwendet worden, um einen bereits bestehenden Regelkatalog schrittweise an ein anderes zu analysierendes System anzupassen. Ein Beispiel für ein solches Vorgehen ist in Abschnitt 7.3 beschrieben.

Die Konsistenz eines Regelkatalogs wird durch die Verwendung des in FUJABA existierenden Mechanismus realisiert. Der Mechanismus erlaubt es, Konsistenzregeln als Story-Pattern zu formulieren, wobei der Zeitpunkt der Überprüfung einer Regel ebenfalls fest-

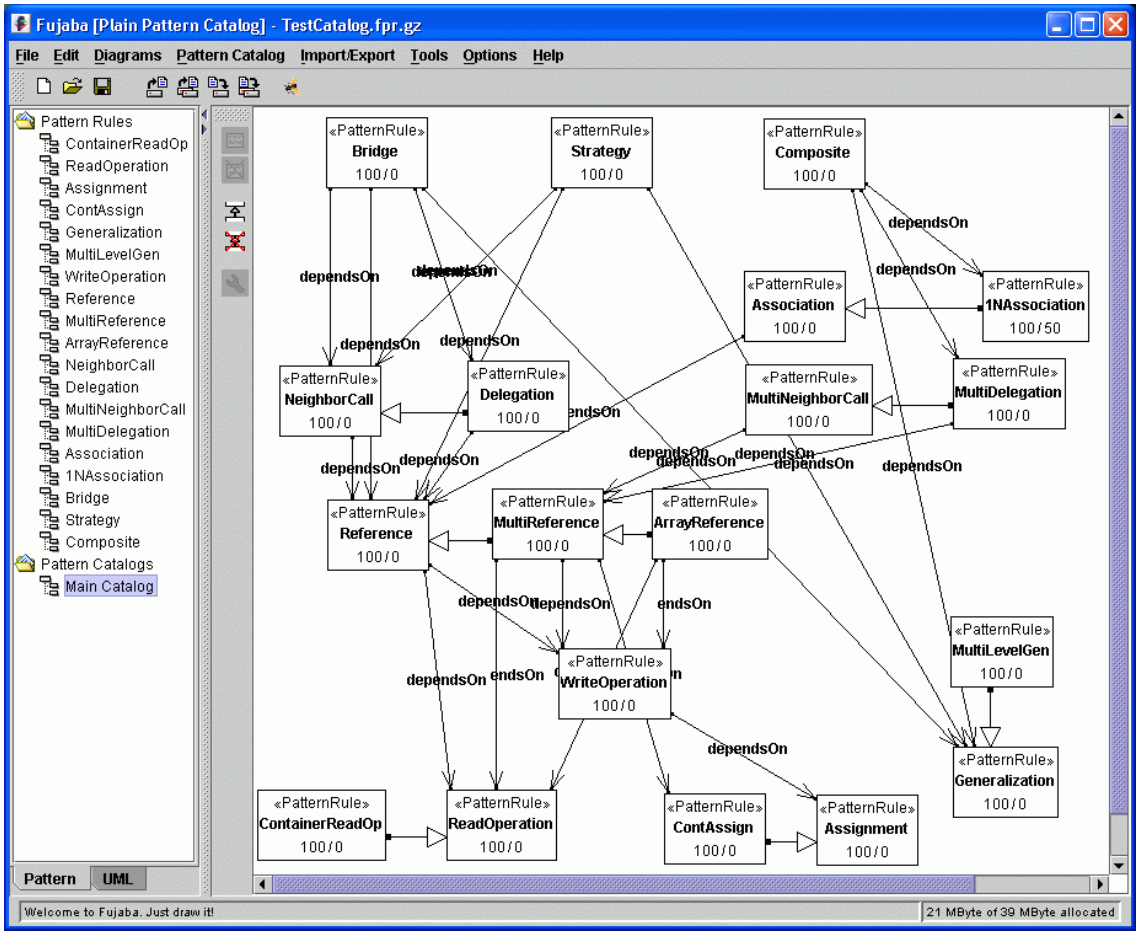


Abbildung 7.2 Regelkatalogbeispiel

gelegt werden kann. Durch die Angabe des Überprüfungszeitpunkts einer Konsistenzregel ist es möglich, sowohl stark restriktive Musterinstanzeingaben wie bei syntaxgesteuerten Editoren [KSW96] als auch freie Eingaben mit nachträglicher Überprüfung zu kombinieren. Der Konsistenzmechanismus ist detailliert in [Wag01, NW01] beschrieben. Im FINITE Prototyp sind die allgemeinen Bedingungen und die Qualifiziererbedingungen für einen Regelkatalog aus Definition 4.10 durch restriktive Dialoge realisiert. Qualifiziererbedingungen werden hingegen nur nach Aufforderung vom Reengineer und vor der automatischen Analyse überprüft.

Inkrementeller Parser

Bevor die automatische Analyse - der Inferenzalgorithmus - starten kann, muss der Quelltext in einen abstraken Syntaxbaum überführt werden. Dies geschieht durch einen Parser [ASU86]. Bei der Entwicklung großer Softwaresysteme werden häufig unterschiedliche Programmiersprachen für die Implementierung verschiedener Teile des Systems eingesetzt. Dementsprechend müssen verschiedene Parser verwendet werden. Das größere Problem in der Praxis bei der Analyse großer Softwaresysteme ist allerdings, dass das Parsen des gesamten Quelltextes vor dem ersten Lauf der automatischen Analyse eine hohe Laufzeit bedeutet. Als Lösung dieses Problems kann ein inkrementeller Parser verwendet werden. Durch den inkrementellen Parser werden jeweils nur die Quelltextteile geparst, die bei der automatischen Analyse benötigt werden.

Im FINITE Prototyp wird ein inkrementeller Parser für Java Quelltexte verwendet. Der Parser ist mit Hilfe des Parsergenerators JavaCC [JCC] generiert worden. Die Erzeugung der Schemainformation aus der Java-Grammatik ist durch den Java Tree Builder erfolgt [JTB]. Die Erzeugung der Schemainformation ist vergleichbar mit der in Kapitel 4.2.1.

Beim inkrementellen Parsen werden zuerst nur die Signaturen von Klassen, Attributen und Methoden in den abstrakten Syntaxbaum übersetzt. Methodenrümpfe werden erst dann in einen abstrakten Syntaxbaum überführt, wenn diese bei der automatischen Analyse benötigt werden.

Anzeige der Analyseergebnisse

Die Anzeige der Analyseergebnisse kann prinzipiell auf verschiedene Arten erfolgen. Für die Erkennung von Entwurfsmusterinstanzen bieten sich UML-Klassendiagramme an, in denen die gefundenen Entwurfsmusterinstanzen durch Annotationen dargestellt werden. Zusätzlich zu den gefundenen Entwurfsmusterinstanzen ist es aufgrund des inkrementellen Ansatzes notwendig, zusätzlich die Musterinstanzen darzustellen, die keinem Entwurfsmuster sondern Teilmuster entsprechen. Diese Musterinstanzen können ebenfalls als Annotationen im Klassendiagramm angezeigt werden, siehe Abbildung 7.3.

Aufgrund der teilweise hohen Anzahl gefundener Musterinstanzen ist die gleichzeitige Darstellung aller Annotationen zur Beurteilung durch den Reengineer ineffektiv. Der Reengineer sollte die Möglichkeit haben, sich die Analyseergebnisse interaktiv anzusehen

und zu beurteilen. Durch die Interaktivität kann der Reengineer einzelne Ergebnisse - gefundene Entwurfsmusterinstanzen - eruieren und zum Beispiel die zugehörigen Annotationen oder die entsprechenden Quelltextausschnitte beurteilen.

Der FINITE Prototyp verwendet für die Anzeige der Analyseergebnisse die in FUJABA vorhandenen Klassendiagramme. Die Möglichkeit, Entwurfsmusterinstanzen durch Annotationen im Klassendiagramm darzustellen, ist ebenfalls vorhanden.

Zusätzlich zu den bereits vorhandenen Klassendiagrammen beinhaltet FUJABA Sichten, die es erlauben, interaktiv oder automatisch eine Sicht für ein Diagramm mit Hilfe von Filtern zu erstellen. Die Filter werden ähnlich zum Konsistenzmechanismus durch Story-Pattern beschrieben. Bei einer automatischen Erzeugung einer Sicht werden alle Diagrammelemente in die Sicht übernommen, die durch den Filter beschrieben werden.

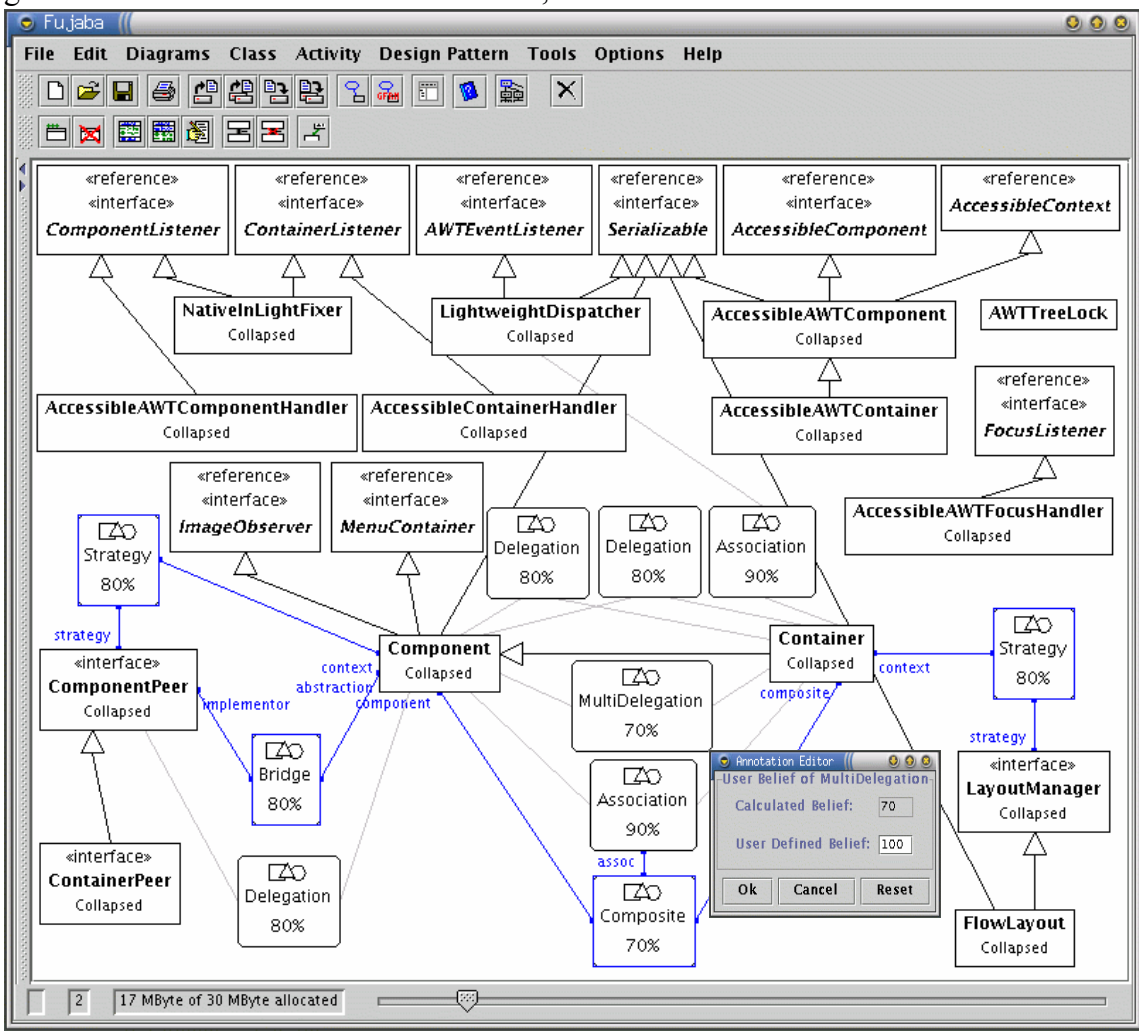


Abbildung 7.3 Anzeige der Analyseergebnisse

Alternativ zur automatischen Erzeugung kann eine Sicht interaktiv durch den Reengineer erstellt werden. Der Reengineer hat dabei die Möglichkeit, einzelne Filter auf Objekte anzuwenden. Die ausgewählten Objekte markieren dabei einen Ausschnitt des annotierten abstrakten Syntaxgraphen, ähnlich wie bei der Angabe eines Kontextes für eine Graphtransformationsregel. Eine detaillierte Beschreibung der Sichten und Filter ist in [Rec01] zu finden.

7.2 Automatische Analyse

Die Anforderungen an die automatische Analyse sind zum Einen ihre Skalierbarkeit, damit der Ansatz für große Systeme eingesetzt werden kann, und zum Anderen die frühzeitige Produktion von Annotationen, die Entwurfsmusterinstanzen entsprechen, damit der Reengineer den verwendeten Regelkatalog gegebenenfalls frühzeitig anpassen kann. Bei der Evaluierung der beiden Anforderungen ist im Wesentlichen die Abstract Windowing Toolkit Bibliothek (AWT) [AWT] analysiert worden.

Die AWT-Bibliothek wird für die Entwicklung graphischer Benutzungsoberflächen verwendet und stellt graphische Komponenten, wie zum Beispiel Fenster, Dialoge, Menüs, etc. zur Verfügung. Einzelne Komponenten werden durch Klassen repräsentiert, und die AWT-Bibliothek verfügt zusätzlich über eine Reihe von Hilfsklassen, die von den Klassen der graphischen Komponenten benutzt werden. Insgesamt umfasst die Bibliothek 429 Klassen in 313 Dateien mit ca. 114.000 Zeilen Quelltext. Die Klassen sind verschiedenen Paketen zugeordnet, wobei die Klassen untereinander durch Beziehungen stark von einander abhängen, so dass eine Analyse der einzelnen Pakete unzureichende Ergebnisse liefern würde.

Bei der Entwicklung der AWT-Bibliothek sind Gamma-Muster verwendet worden, obwohl sie teilweise weder im Quelltext noch in Entwurfsdokumenten angegeben werden. Ein Überblick der in der Bibliothek vorhandenen Gamma-Muster ist in [Gra97] veröffentlicht. Zum Beispiel wird für die Verwaltung der Objekte, an die Ereignisse geschickt werden, das Gamma-Muster Observer verwendet und bei der Ereignisbehandlung wird das Gamma-Muster ChainOfResponsibility verwendet.

Der initiale Regelkatalog, der bei der Analyse der AWT-Bibliothek verwendet worden ist, ist im Anhang aufgeführt. Die gefundenen Entwurfsmusterinstanzen bei der automatischen Analyse entsprechen den Ergebnissen von Seemann und von Gutenberg [SvG98]. Der Regelkatalog ist dabei so angepasst worden, dass keine false-positives mehr produziert worden sind, siehe [Wen01].

Eine automatische Analyse der AWT-Bibliothek hat nach etwa 15 Sekunden die erste Annotation produziert, die einem Entwurfsmuster entsprach. Insgesamt hat die Analyse etwa 22 Minuten auf einem zur Zeit handelsüblichen Rechner mit ca. 2 GHz Prozessortakt und 1 GB Hauptspeicher gedauert. Inklusive Anpassung des Regelkatalog hat sich eine Gesamtanalysezeit von etwa einem Tag ergeben.

Der Regelkatalog ist an die Java Beans Programmierrichtlinien angepasst, wodurch er auch für andere Systeme, die nach den gleichen Richtlinien implementiert worden sind, als Ausgangsregelkatalog verwendet werden kann. Die automatische Analyse der Java Generic Library [JGL] - eine Bibliothek von Containerdatenstrukturen und etwa 53.000 Zeilen Quelltext - dauert etwa 1,5 Minuten und produziert nach ca. 2 Sekunden die erste Annotation, die einer Entwurfsmusterinstanz entspricht. Eine manuelle Prüfung der Ergebnisse hat allerdings ergeben, dass ca 25% der Ergebnisse false-positives waren. Eine Anpassung des Regelkatalogs zur Produktion weniger false-positives ist daher notwendig. Ebenso hat eine automatische Analyse des Open Source Webservers JigSaw [W3C] mit etwa 151.000 Zeilen Quelltext ca. 35 Minuten gedauert, wobei die erste Entwurfsmusterinstanz bereits nach ca. 10 Sekunden erkannt worden war. Das zur Zeit größte analysierte System ist die Entwicklungsumgebung FUJABA mit etwa 750.000 Zeilen Quelltext. Die automatische Analyse hat 5-6 Stunden gedauert. Die Analyse war allerdings nur mit einem Hauptspeicher von mehreren GB möglich, damit die Auslagerung von Speicherbereichen gering gehalten wurde. Optimierte Speicherstrategien würden den Prototypen an dieser Stelle wesentlich verbessern.

Die Dauer einer automatischen Analyse ist nicht nur von der zu analysierenden Systemgröße abhängig, sondern auch von der Anzahl Regeln im verwendeten Regelkatalog und den Abhängigkeiten und den Verfeinerungsbeziehungen der Regeln. Wird zum Beispiel die Delegation-Regel und die MultiDelegation-Regel im Regelkatalog weggelassen, so verkürzt sich die Analysezeit in den obigen Beispielen um ca. 40%. Dies liegt zum Einen an der Komplexität der Musterinstanzregeln und zum Anderen daran, dass alle Musterinstanzregeln, die Entwurfsmustern entsprechen, von ihnen abhängen.

Die obigen Beispiele haben gezeigt, dass die automatische Analyse frühzeitig relevante Ergebnisse produziert, die für eine frühzeitige Anpassung eines Regelkatalogs an ein zu analysierendes System verwendet werden können. Im Folgenden Abschnitt wird ein Beispielvorgehen für die Anpassung eines Regelkatalogs zur Erkennung von Entwurfsmusterinstanzen an ein zu analysierendes System vorgestellt.

7.3 Beispiel für ein Analysevorgehen

Die Analyse eines Softwaresystems mit dem in den vorherigen Kapiteln vorgestellten Ansatz beginnt mit der Auswahl eines Regelkatalogs, der bereits für die Analyse eines anderen Systems verwendet und angepasst worden ist. Die Regeln des ausgewählten Regelkatalogs sollten dabei nicht derart angepasst sein, dass sie beim neu zu analysierenden System keine oder nur wenige Annotationen erzeugen. Um dies zu vermeiden, sollte der Reengineer vor der Auswahl des Regelkatalogs Informationen über Programmierrichtlinien oder verwendete Bibliotheken einholen. Sollten solche Informationen nicht vorhanden sein, bleibt nur ein Blick in den Quelltext, um Programmierstile oder -richtlinien manuell zu extrahieren. Besonders geeignet sind dafür Implementierungen von As-

soziationen oder bei nicht objektorientierten Programmiersprachen Klassenstrukturen [Har00].

Assoziationen kommen in der Struktur fast jedes Entwurfsmusters beziehungsweise Gamma-Musters vor und verbinden Klassen miteinander. Allerdings besitzen gängige objektorientierte Programmiersprachen im Gegensatz zu einem Sprachkonstrukt für eine Klasse kein Sprachkonstrukt für eine Assoziation.

In der Struktur von Entwurfsmustern werden eine Reihe verschiedener Assoziationstypen verwendet. Zum Beispiel unterscheidet die UML bi-direktionale, gerichtete, sortierte, geordnete, mehrwertige und qualifizierte Assoziationen. Hinzu kommen die Kardinalitäten an den - mindestens zwei - Rollen der Assoziation.

Gängige objektorientierte Programmiersprachen verwenden Zeiger für eine Beziehung zwischen zwei Klassen. Ein Zeiger ist eine Implementierung einer gerichteten Assoziation, deren Kardinalität 1 ist, und wird auch als Referenz bezeichnet. Der folgende Java Quelltext zeigt drei mögliche Implementierungen für gerichtete Assoziationen zwischen der Klasse `House` und `Elevator` beziehungsweise `Level`.

```
1: public class House {
2:    // toOne reference to Elevator
3:         public Elevator myElevator;
4:    // toOne reference to Level
5:         private Level curLevel;
6:         public Level getCurLevel() {
7:           return curLevel; } // end getLevel
8:         public void setLevel(Level l) {
9:           if (curLevel != l)
10:            curLevel = l; } // end setLevel
11:   // toMany reference to Level
12:        private TreeSet levels;
13:        public void addToLevels (Level l) {
14:          if (!levels.contains (l)) {
15:            levels.add (l);
16:            l.setHouse (this);}
17:        } // end addToLevels
18:}
```

Abbildung 7.4 Quelltextbeispiel für Referenzbeziehungen

Die Variable `myElevator` vom Typ `Elevator` in den Zeilen 2-3 ist dabei die einfachste Implementierung einer Referenz. Da die Variable von anderen Objekten modifiziert werden, kann ist diese Implementierung aus softwaretechnischer Sicht nicht gut geeignet. Sie findet sich insbesondere in älteren Softwaresystemen und in Systemen, in denen der Speicherplatz stark begrenzt ist, wie zum Beispiel eingebettete Systeme.

Eine aus softwaretechnischer Sicht bessere Implementierung einer Referenz ist in den Zeilen 5-10 dargestellt. Die Variable `curLevel` kann von einem anderen Objekt nur

durch die Methode `getCurLevel` gelesen oder durch `setCurLevel` gesetzt werden. Diese Implementierung entspricht den JavaBeans Programmierrichtlinien [Bea97, Pra97] und wird für den Zugriff aller Variablen einer Klasse genutzt.

Die vorherigen Beispielimplementierungen sind alles Implementierungen von Referenzen gerichteter Assoziationen mit Kardinalität 1. Für die Implementierung anderer Kardinalitäten werden Arrays oder Containerklassen verwendet [FNTZ98]. Beispielsweise stellt Java eine Bibliothek von Containerklassen, die Java Foundation Classes (JFC), zur Verfügung. Die Containerklasse `TreeSet` wird im dritten Beispiel in den Zeilen 12-17 verwendet, um eine bi-direktionale Assoziation zwischen der Klasse `House` und der Klasse `Level` zu implementieren. Dass es sich um eine Assoziation zwischen `House` und `Level` handelt, ist an dem Parametertyp `Level` des Parameters `l` der Methode `addToLevels` erkenntlich (Zeile 13). Das Objekt wird, wenn es noch nicht im Container vorhanden ist (Zeile 14), hinzugefügt (Zeile 15). In diesem Fall ist es notwendig, die Schnittstellen der verwendeten Containerklassen zu kennen. Das es sich um eine bi-direktionale Assoziation handelt deutet der Aufruf der Methode `setHouse` auf dem als Parameter übergebenen Objekt hin. Für eine genaue Aussage muss die Implementierung der setHouse-Methode untersucht werden. Derartige Implementierungen bi-direktionaler Assoziationen zwischen Klassen werden von CASE-Werkzeugen generiert. Für die Quelltextgenerierung anderer Assoziationstypen werden entsprechende Containerklassen verwendet. Der generierte Quelltext wird als Ausgangspunkt für ein zu implementierendes Softwaresystem verwendet.

Für die Erkennung der obigen Beispielimplementierungen mit dem hier vorgestellten Ansatz reichen fünf Musterinstanzregeln aus, siehe [NWW03]. Diese fünf Regeln sind in mehreren Prozessiterationen aus der Musterinstanzregel in Abbildung 7.5 entstanden. Die Musterinstanzregel annotiert alle Variablen, deren Typ eine Klasse sind. Für die ersten beiden oberen Implementierungsbeispiele ist diese Regel ausreichend, allerdings wird im dritten Fall nicht die Klasse `House` und `Level`, sondern die Klasse `House` und `TreeSet` annotiert. Es handelt sich also um false-positives.Trotzdem hat sich die dargestellte Musterinstanzregel als guter Ausgangspunkt erwiesen.

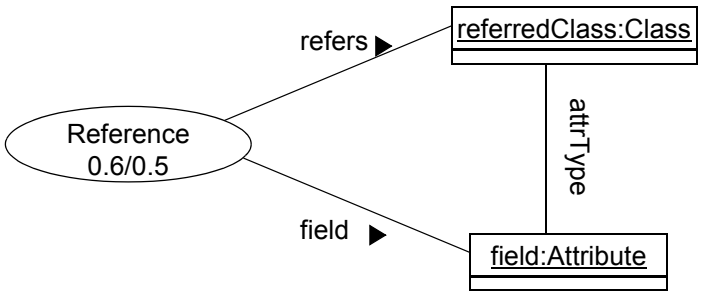


Abbildung 7.5 Musterinstanzregelbeispiel für Referenzen

Im Anschluss an die Anpassung der Musterinstanzregeln für Beziehungen zwischen Klassen sollte der Regelkatalog für die Erkennung einer Entwurfsmusterinstanz angepasst werden. Als geeignet haben sich dafür die Entwurfsmuster Composite, Strategy oder Bridge herausgestellt, da sie sowohl statische als auch dynamische Anteile in ausreichendem Maße enthalten. Die Musterinstanzregeln für diese drei Entwurfsmuster sind dabei weitgehend unabhängig von der konkreten Implementierung. Sie bauen auf anderen Musterinstanzregeln auf, siehe Regelkatalog im Anhang. Die Anpassung eines Regelkatalogs für die Erkennung eines der Entwurfsmuster besteht also im Wesentlichen aus der Anpassung der Hilfsregeln. Musterinstanzregeln für weitere Entwurfsmuster können anschließend basierend auf diesen Hilfsregeln aufgebaut werden.

Der im Anhang vorhandene Regelkatalog ist auf diese Art und Weise entstanden, wobei mit der Anpassung des Regelkatalogs zur Erkennung des Composite-Musters begonnen worden ist. Für die Erkennung des Bridge- und Strategy-Musters konnten die angepassten Hilfsregeln für die Erkennung des Composite-Musters ohne weitere Änderungen übernommen worden.

7.4 Genauigkeits- und Schwellwertadaption

Die hier vorgestellte automatische Adaption der Genauigkeits- und Schwellwerte von Musterinstanzregeln ist das Ergebnis aktueller Forschungsarbeit und ist noch nicht in den FINITE Prototypen integriert. Es existieren daher noch keine statistisch fundierten Daten, die Aussagen über die Verbesserung der Analyse durch die Adaption zulassen.

Das vorgestellte Adaptionsverfahren ist allerdings prototypisch implementiert worden. Die Größe des simulierten Fuzzy-Petrinetzes hat etwa einer Analyse eines 100.000 Zeilen großen Quelltext Systems entsprochen. Die Korrekturen des Reengineer sind durch einen Zufallszahlengenerator simuliert worden, und die Berechnung der Adaptionswerte ist jeweils direkt nach einer Korrektur erfolgt.

Nach jeweils 1.000 Korrekturen sind die neu berechneten Präzisionswerte in die Berechnung der Genauigkeitswerte übernommen worden. Der Zeitaufwand für die Berechnung der Genauigkeitswerte war dabei höher als die Berechnung der Präzisionswerte der 1.000 vorgenommenen Korrekturen. Im Vergleich mit einem Neuronalen-Netz, das als Referenzimplementierung verwendet worden ist, kann die Zeit für die Berechnung der Adaptionswerte vernachlässigt werden. Im Gegensatz dazu hat das Neuronale-Netz teilweise mehrere Minuten benötigt, um eine Korrektur zu verarbeiten. Dies liegt in erster Linie an der hohen Anzahl von Lernaufgaben, die aufgrund einer Korrektur eines Werts erzeugt werden.

Im Vergleich zu einem Neuronalen-Netz hat das hier vorgestellte statistische Verfahren allerdings die Eigenschaft, sich schlechter von einem lokalen Minimum auf das globale Minimum zu bewegen. Folglich sollte die automatische Adaption der Genauigkeits- und Schwellwerte erst dann erfolgen, wenn der verwendete Regelkatalog sich nicht mehr von einer Iteration auf die nächste Iteration des Analyseprozesses grundlegend ändert. Solche

grundlegenden Änderungen erfolgen in der Regel in den ersten Iterationsschritten. In späteren Iterationsschritten unterliegen Musterinstanzregeln lediglich kleinen Änderungen.

Für die Adaption der Genauigkeits- und Schwellwerte von Musterinstanzregeln ist eine Schnittstelle entwickelt worden, wodurch ein einfaches Austauschen eines konkreten Adaptionsverfahrens ermöglicht wird. Anstelle des vorgestellten statistischen Verfahrens können zum Beispiel Neuronale-Netze oder probabilistische verwendet werden. Durch die Schnittstelle kann der FINITE Prototyp als eine Art Testumgebung für verschiedene Adaptionsverfahren verwendet werden.

KAPITEL 8: ZUSAMMENFASSUNG UND AUSBLICK

Entwurfsmuster beschreiben Lösungen immer wiederkehrender Probleme und sind das Ergebnis der Analysen bestehender Softwaresysteme. Bei der Dokumentationsrückgewinnung für ein Softwaresystem können Entwurfsmuster für die Erkennung von Systemgrenzen oder Modulen genauso verwendet werden wie zur Erkennung von Kommunikation, wie zum Beispiel Protokollen.

Die Beschreibung eines Entwurfsmusters erfolgt informal durch Prosa, wodurch sich eine hohe Anzahl von Implementierungsvarianten ergibt. Eine konkrete Implementierung eines Entwurfsmusters entspricht einer Entwurfsmusterinstanz. Die hohe Anzahl Implementierungsvarianten lässt eine automatische Erkennung aller Entwurfsmusterinstanzen fehlschlagen. Allerdings werden in einem Softwaresystem nicht alle Implementierungsvarianten verwendet. Daher ist in dieser Arbeit eine inkrementelle Entwurfsmustererkennung vorgestellt worden. In einem semi-automatischen Analyseprozess passt der Reengineer einen Regelkatalog, dessen Regeln Entwurfsmusterinstanzen entsprechen, so an, dass dieser nur Regeln enthält, die im Softwaresystem vorhandene Entwurfsmuster erkennen. Durch den semi-automatischen Analyseprozess kann der Reengineer zusätzlich Hypothesen und weitere Informationen, zum Beispiel vorhandene Teildokumentationen, mit in die Analyse einbringen.

Zur Formalisierung der Entwurfsmusterinstanzen sind Graphtransformationsregeln in Kombination mit Fuzzylogik verwendet worden. Jede Musterinstanzregel besitzt einen Genauigkeitswert, der der relativen Anzahl false-positives entspricht, die die Musterinstanzregel für ein zu analysierendes System produziert. Die Genauigkeitswerte werden zur Berechnung der Präzisionswerte der produzierten Ergebnisse verwendet. Die Präzisionswerte dienen dem Reengineer bei der Beurteilung der Ergebnisse. Ergebnisse mit niedrigem Präzisionswert sind im Allgemeinen false-positives.

Für die automatische Analyse, bei der die Musterinstanzregeln eines Regelkatalogs auf einen abstrakten Syntaxgraphen des zu analysierenden Quelltextes angewendet werden, ist ein Inferenzalgorithmus entwickelt worden, der relevante Ergebnisse frühzeitig produziert. Relevante Ergebnisse sind zum Beispiel erkannte Entwurfsmusterinstanzen. Insbesondere bei der Analyse großer Systeme ist die frühzeitige Produktion von Ergebnissen notwendig, damit der Reengineer den weiteren Verlauf der Analyse gegebenenfalls frühzeitig beeinflussen kann.

Allgemein besteht bei fuzzy-basierten Ansätzen - wie in dieser Arbeit - das Problem, den Fuzzywert, in diesem Fall den Genauigkeitswert für eine Musterinstanzregel, zu bestim-

men. Der Genauigkeitswert ist abhängig vom zu analysierenden System und muss vom Reengineer initial geschätzt werden. Während der Analyse des Systems werden die Korrekturen des Reengineer an Präzisionswerten der Ergebnisse protokolliert, und durch ein statistisches Verfahren werden die Genauigkeitswerte der Musterinstanzregeln angepasst. Im Laufe einer Analyse eines Systems nähern sich die Genauigkeitswerte so an die relative Anzahl produzierter false-positives einer Musterinstanzregel an.

Der vorgestellte Ansatz zur Erkennung von Entwurfsmusterinstanzen in großen Systemen ist im FINITE Prototyp als Teil und basierend auf der FUJABA Entwicklungsumgebung implementiert worden. Neben einem Editor für Regelkataloge und Musterinstanzregeln sind Klassendiagramme zur Anzeige der Analyseergebnisse verwendet worden. Ein flexibler Filtermechanismus erlaubt es dem Reengineer, sich Teilergebnisse anzusehen oder Ergebnisse mit niedrigen Präzisionswerten auszublenden. Der praktische Einsatz des FINITE Prototypen hat gezeigt, dass der vorgestellte Ansatz für die Erkennung von Entwurfsmusterinstanzen, insbesondere hinsichtlich der zu analysierenden Systemgröße, die existierenden Ansätze übertrifft.

Bei der Evaluierung hat sich herausgestellt, dass der Prototyp insbesondere in seiner graphischen Benutzungsoberfläche verbessert werden kann. Zusätzlich haben sich Optimierungsmöglichkeiten beim Inferenzalgorithmus ergeben, die die Laufzeit der automatischen Analyse verkleinern. Dies sind unter anderem unterschiedliche Rangnummern für eine Musterinstanzregel im top-down und bottom-up Modus und eine verbesserte Erkennung, ob eine Regel auf einem gegebenen Kontext bereits angewendet worden ist.

Im top-down Modus wird versucht, alle möglichen Regelanwendungen auf einen Kontext auszuführen. Bei Regelkatalogen mit hohen Regelrängen ergibt sich allgemein eine hohe Anzahl möglicher Regelanwendungen. Eine Verbesserung könnte erreicht werden, indem die Quelltextteile, in der eine Entwurfsmusterinstanz vermutet wird, gezielt bestimmt werden. Auf diesen Quelltextanteilen wären weniger mögliche Regelanwendungen auszuführen.

Ein weiteres Anwendungsgebiet, in dem der hier vorgestellte Ansatz angewendet wird, ist das Reengineering föderierter Datenbanken. Im Gegensatz zu Datenbanksystemen, in denen Abhängigkeiten und Integritätsbedingungen explizit formuliert werden können, erfolgt dies bei föderierten Datenbanken in den zugehörigen Applikationen. Dabei können sie über mehrere Applikationen verstreut sein oder sich innerhalb einer Applikation nur auf bestimmte Teile der Datenbank beziehen. Bei der Analyse werden in einem ersten Schritt die Schemainformationen der beteiligten Datenbanken mit dem VARLET Ansatz extrahiert. In einem zweiten Schritt werden die Applikationen mit dem hier vorgestellten Ansatz analysiert, wobei die extrahierten Schemainformationen als Orientierungshilfe für den Reengineer dienen. Ziel ist es, verschiedene Abhängigkeiten zwischen den Datenbanken zu extrahieren. Dazu sind in der Arbeit verschiedene Abhängigkeitstypen entwickelt worden, siehe auch [WNGJ02]. Aufbauend auf diesen Informationen wird in einem wei-

teren Schritt das Schema der föderierten Datenbank mit seinen Abhängigkeiten und Integritätsbedingungen erstellt.

Um weitere Anwendungsgebiete zu erschließen, besteht eine Erweiterungsmöglichkeit des Ansatzes darin, Musterinstanzregeln zusätzlich zu ihrem Genauigkeitswert weitere Attributwerte zuzuordnen. Zum Beispiel könnten dadurch speichereffiziente oder laufzeiteffiziente Implementierungsvarianten eines Entwurfsmusters unterschieden werden. Bei der Verwendung von Mustersprachen werden zusätzlich zu den Mustern auch Bewertungen bezüglich der Kombinationen von Mustern beziehungsweise Musterinstanzen beschrieben. Diese Bewertungen können durch eine Attributierung der Musterinstanzregeln in die Ergebnisse der Analyse eines Softwaresystems einfließen und zusätzlich Informationen über das System liefern. Zum Beispiel können damit Unverträglichkeiten von Musterinstanzen erkannt werden.

Aufbauend auf der Erkennung von Unverträglichkeiten könnte der hier vorgestellte Ansatz um die Möglichkeit der Ersetzung einer Musterinstanz durch eine andere Implementierungsvariante ergänzt werden. Dies entspräche dem in [JZ97] vorgestellten Ansatz zur Ersetzung “schlechter” Implementierungsvarianten durch “bessere” Implementierungsvarianten eines Entwurfsmusters. Für eine Umsetzung wäre eine detaillierte Analyse des Quelltextes notwendig, damit die Ersetzung nicht zu einem nicht funktionierenden oder fehlerhaften System führt. Für große Softwaresysteme ist eine solch detaillierte Analyse im Allgemeinen nicht durchführbar, allerdings könnte eine Ersetzung weniger komplexer Muster wie zum Beispiel Implementierungsmustern für Elemente eines Softwaredesigns erfolgreich umgesetzt werden.

Der Fokus für zukünftige Arbeiten liegt in der Kombination beziehungsweise Integration des hier vorgestellten Ansatzes zur Erkennung von Entwurfsmusterinstanzen in großen Softwaresystemen mit anderen Reverse-Engineering Ansätzen. Der FINITE Prototyp könnte dabei als Ausgangspunkt für eine Reverse-Engineering Workbench dienen, in denen verschiedene Analysetechniken integriert werden. Ansatzpunkt für eine Kopplung beziehungsweise Integration ist der vorgestellte Analyseprozess, der es erlaubt, Zwischenergebnisse für andere Analysen bereitzustellen sowie Ergebnisse anderer Analysen mit in den Prozess einzubeziehen. Zum Beispiel werden gefundene Entwurfsmusterinstanzen in [Wen03] als Ausgangspunkt für eine dynamische Analyse des Softwaresystems verwendet. Die dynamischen Anteile der gefundenen potenziellen Entwurfsmusterinstanzen werden durch die dynamische Analyse überprüft. Das Ergebnis der Analyse fließt anschließend in den Analyseprozess ein. Durch die dynamische Analyse wird die Anzahl der false-positives im Gegensatz zur hier vorgestellten statischen Analyse des Quelltextes verringert.

ANHANG: REGELKATALOGBEISPIEL

In den folgenden Abbildungen ist ein Regelkatalog dargestellt, bei dem es sich herausgestellt hat, dass er sich als initialer Regelkatalog zur Analyse von Softwaresystemen, die nach den Java Beans Programmierichtlinien implementiert worden sind, eignet. Die Programmierrichtlinien werden vielfach in Open Source Projekten wie den Jakarta Projekten oder SourceForge Projekten eingesetzt.

Verfeinerungen dieses Regelkatalogs sind für die Analyse der Java Windowing Toolkit Bibliothek [AWT] und den JigSaw Webserver [W3C] verwendet worden. Ergebnisse der Analyse sind in [Wen01, NWW01, NWZ03, NZ03] publiziert worden.

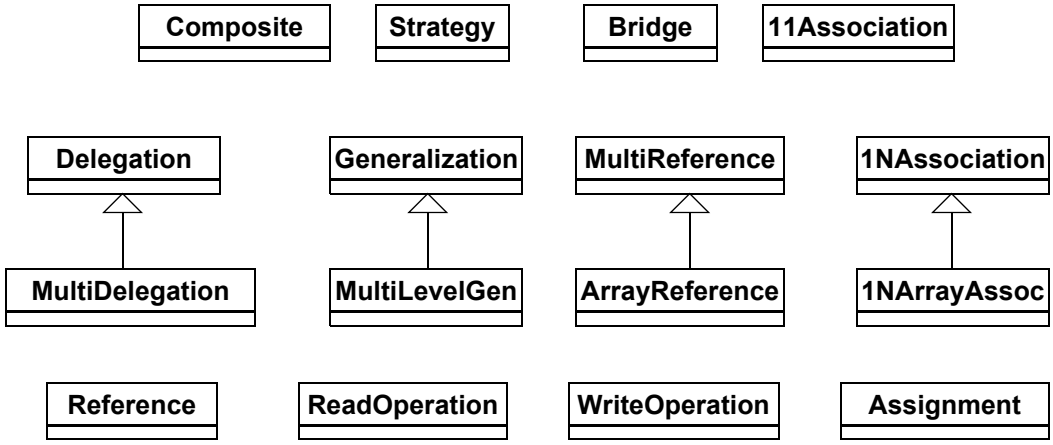


Abbildung A.1 Regeldefinitionsdiagramm für Beispielkatalog

Die Composite-Regel ist bereits für als Beispiel für zur Erläuterung des Inferenzalgorithmus verwendet worden.

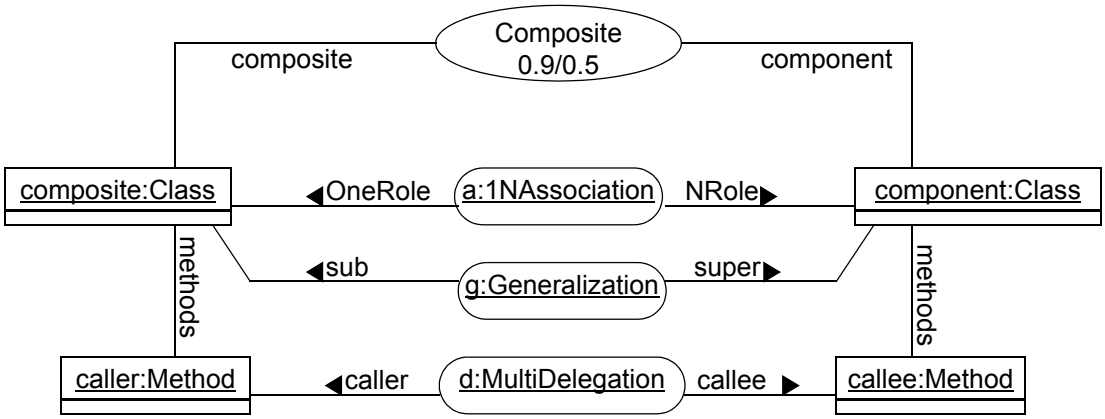


Abbildung A.2 Composite-Regel

Die Bridge-Regel ist bereits als Beispiel in Kapitel 4 dargestellt und dient zur Kopplung von Systemteilen.

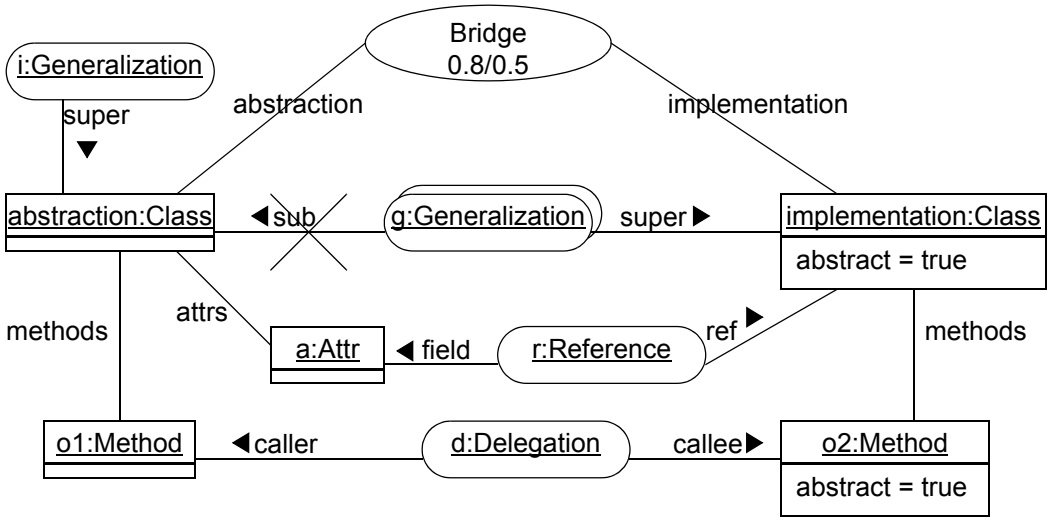


Abbildung A.3 Bridge-Regel

Die Strategy-Regel ist der Bridge-Regel nahezu identisch, obwohl die Intention der beiden Gamma-Muster verschieden ist.

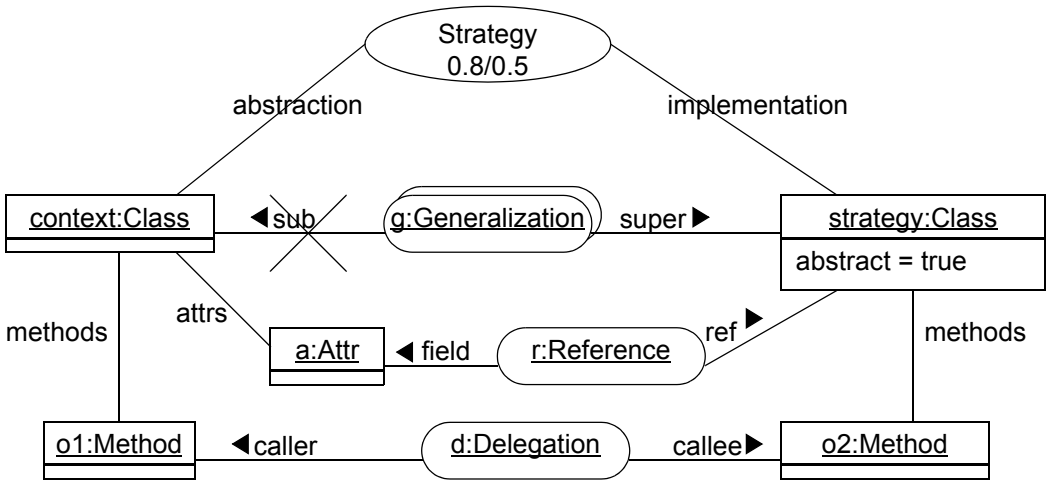


Abbildung A.4 Strategy-Regel

Die 11Association-Regel dient zur Erkennung von Assoziationen zwischen zwei Klassen, dessen Kardinalität auf beiden Seiten 1 ist. Während des Inferenzalgorithmus werden dabei für eine Assoziation zwei Annotaionsknoten erzeugt, weil die beiden Klassen jeweils als left und right Rolle annotiert werden. Außerdem wird für jedes Referenz-Paar eine Annotation erzeugt. Die Genauigkeit ist daher niedrig angesetzt.

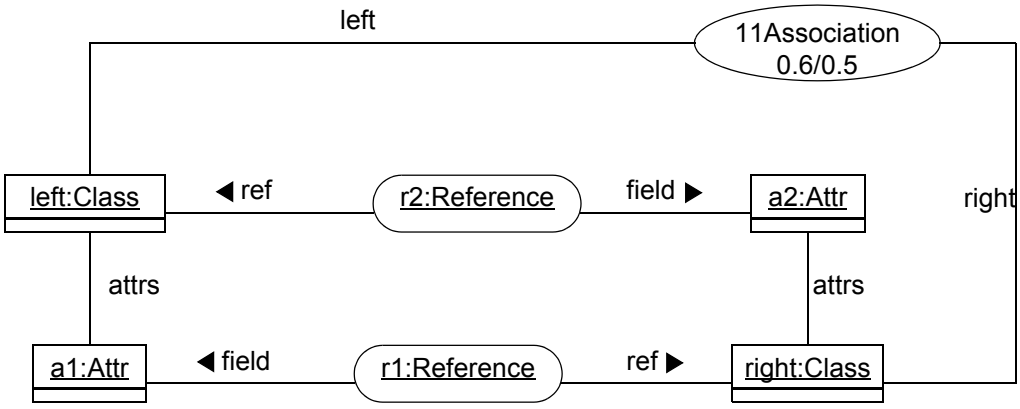


Abbildung A.5 11Association-Regel

Eine Assoziation zwischen zwei Klassen mit Kardinalität 1 auf der einen und unbeschränkter Kardinalität auf der anderen Seite wird durch die 1NAssoziation erkannt.

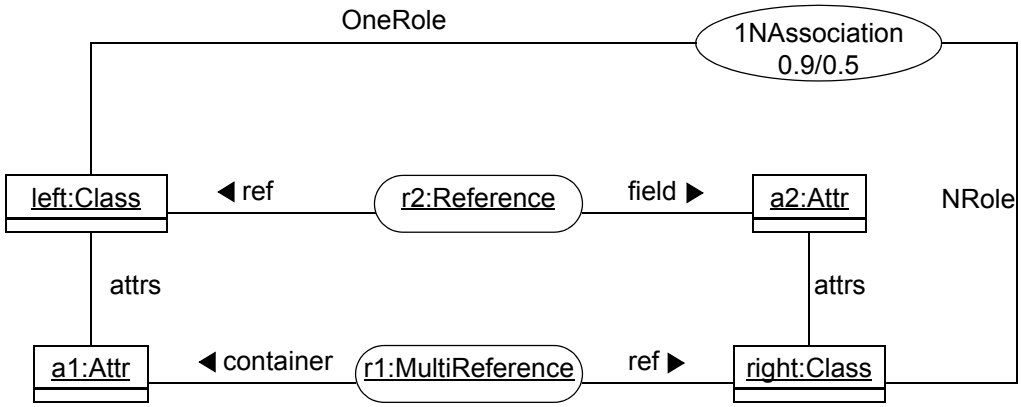


Abbildung A.6 1NAssociation-Regel

1NAssoziationen, die mit Hilfe von Arrays implementiert worden sind werden durch die 1NArrayAssoc-Regel identifiziert.

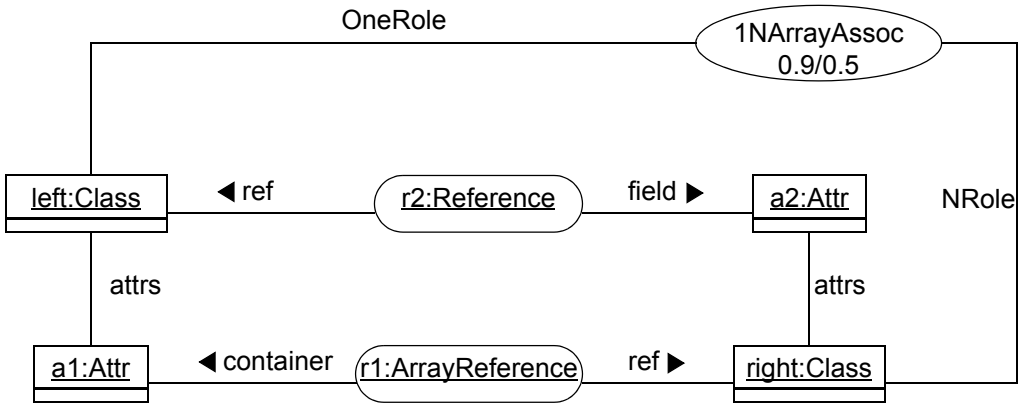


Abbildung A.7 1NArrayAssoc-Regel

Die Delegation-Regel ist bereits als Beispiel bei der Formalisierung von Entwurfsmusterinstanzen in Kapitel 4 vorgestellt worden und identifiziert Methoden, deren Rumpf einen Methodeaufruf enthalten, der eine gleichnamige Methode einer anderen Klasse aufruft..

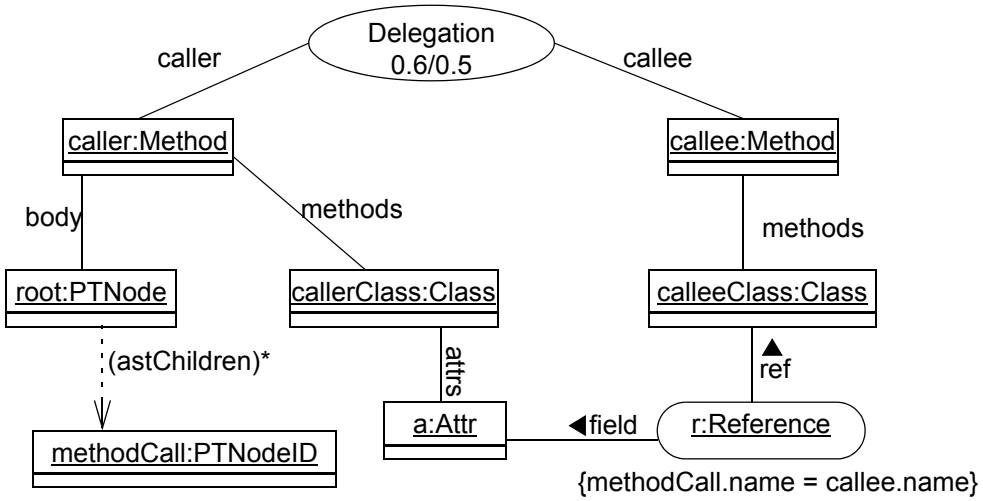


Abbildung A.8 Delegation-Regel

Eine MultiDelegation-Regel ist eine Delegation, bei der die Delegation an mehrere Objekte delegiert wird.

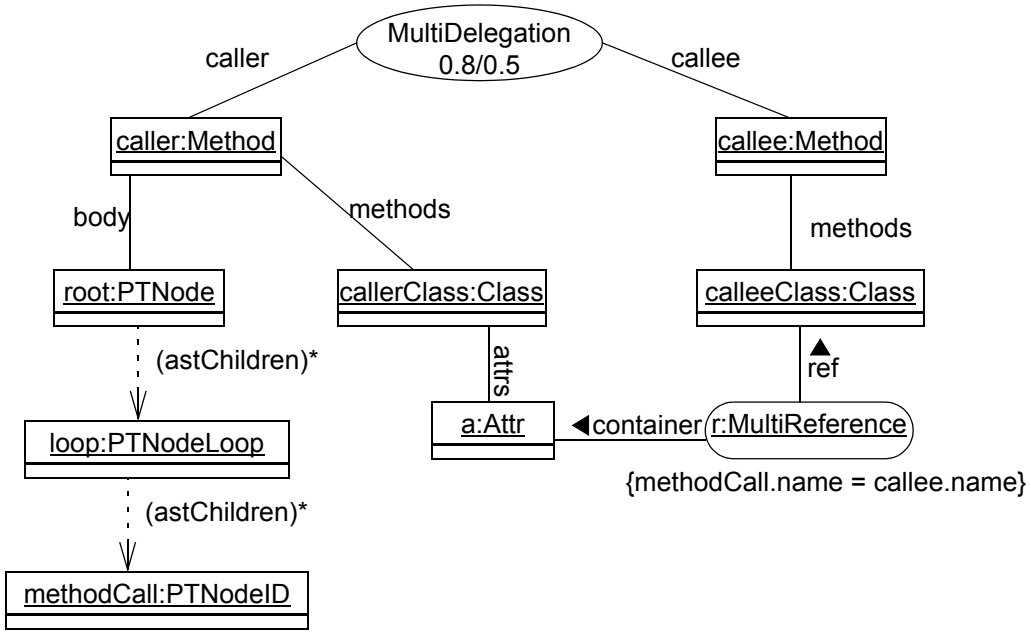


Abbildung A.9 MultiDelegation-Regel

Referenzen werden ebenfalls als Regel definiert, weil die JavaBeans Programmierrichtlinie verlangt, dass diese mit Zugriffsmethoden für den lesenden und schreibenden Zugriff versehen sein müssen.

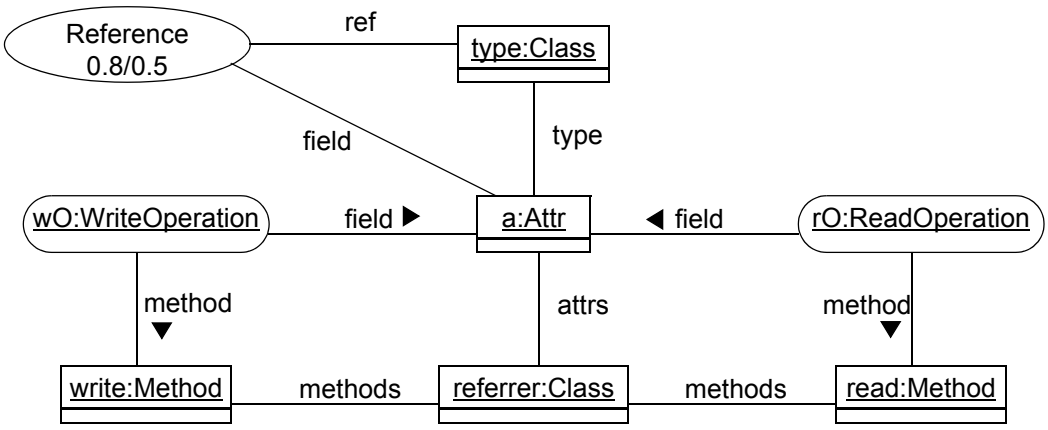


Abbildung A.10 Reference-Regel

Bei einer MultiReferenz enthält eine Methode eine Zuweisung des als Parameter übergebenen Objekts an einen Containter.

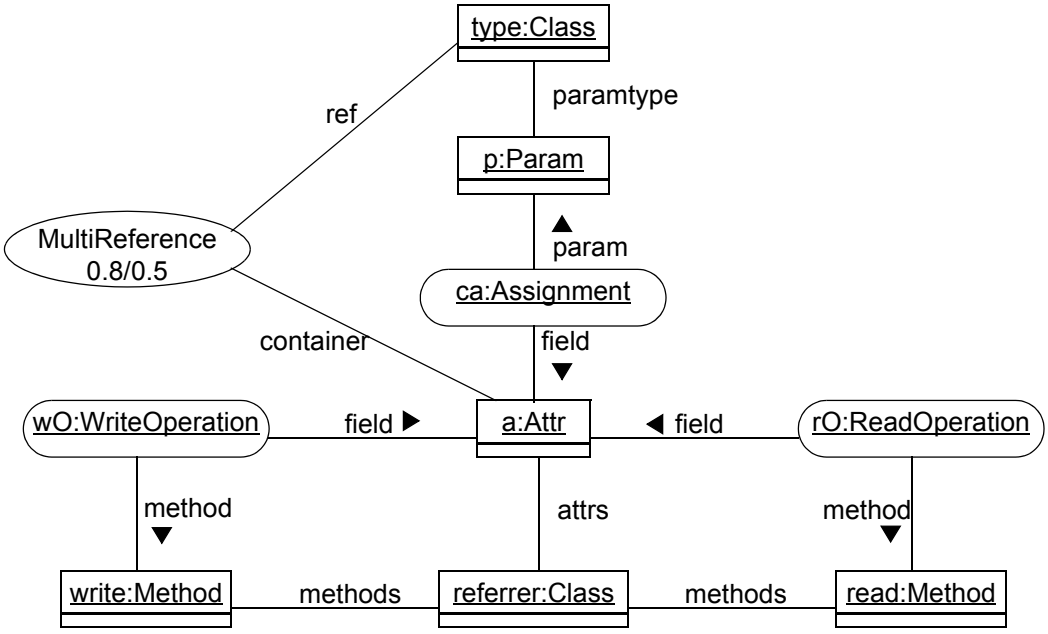


Abbildung A.11 MultiReference-Regel

Eine ArrayReferenz identifiziert MultiReferenzen zwischen zwei Klassen, die als Array implementiert worden sind.

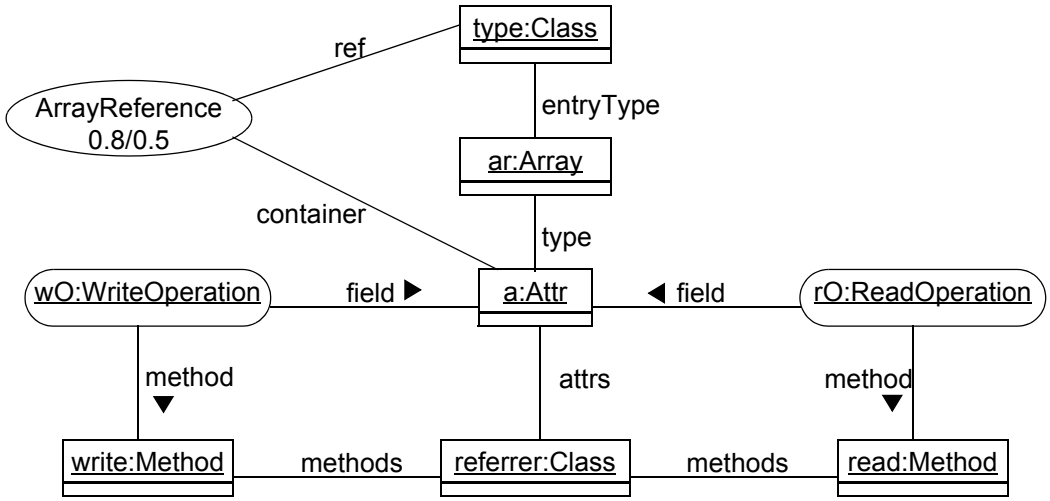
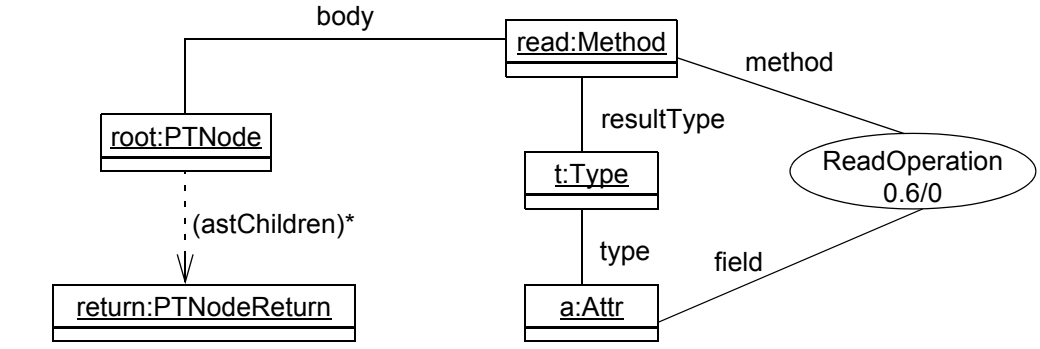


Abbildung A.12 ArrayReference-Regel

Die Read- und WriteOperation-Regel identifizieren Lese- und Schreibzugriffsmethoden auf Attribute mit beliebigen Datentyp. Die Funktion regExp ist eine Methode zur Überprüfung regulärer Ausdrücke. Der erste Parameter ist die Zeichenkette, in der der als zweite Parameter übergebene reguläre Ausdruck gesucht wird. Für JavaBeans Programmierrichtlinien ergeben sich als Namen für Attributzugriffsmethoden get, is, iterator, set und remove.



{regExp(readMethod.name,“get|is|iterator“+a.name) and regExp(return.value,“.*”+a.name)}

Abbildung A.13 ReadOperation-Regel

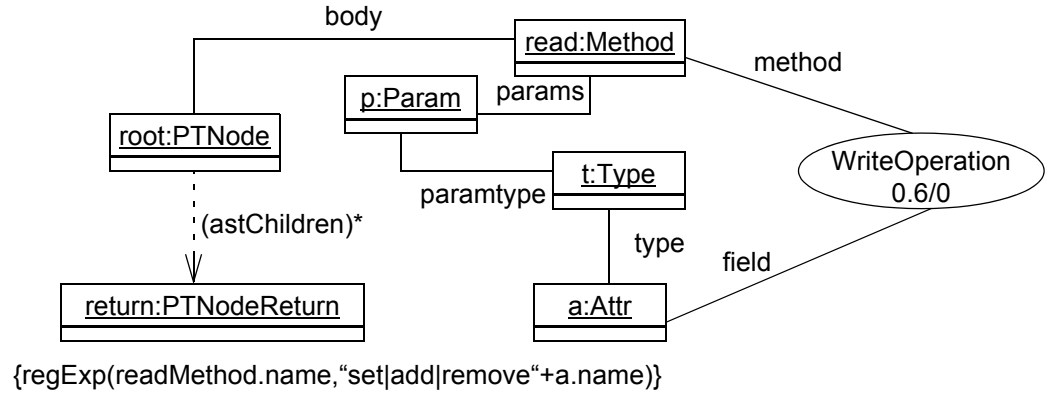


Abbildung A.14 WriteOperation-Regel

Die Assignment-Regel identifiziert Methoden, deren Methodenrumpf eine Zuweisung enthält, bei der einem Attribut einer Klasse ein übergebener Parameter zugewiesen wird.

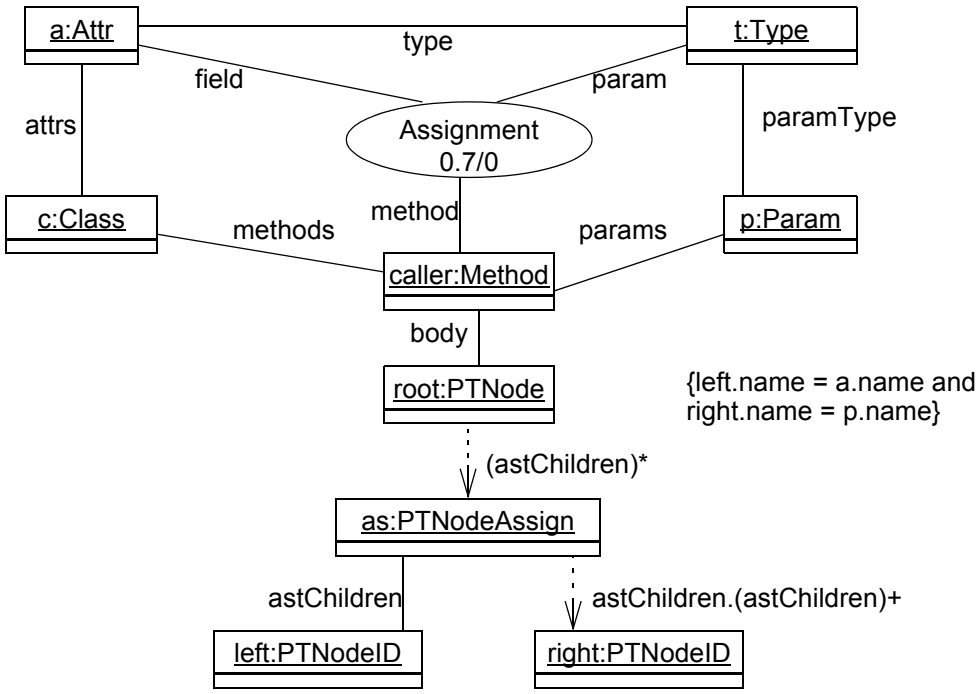


Abbildung A.15 Assignment-Regel

Die MultiLevelGen-Regel und die Generalization-Regel identifizieren - gegebenenfalls rekursiv - Vererbungsstrukturen. Der einheitliche Zugriff ist durch die Verfeinerungskante im Regeldefinitionsdiagramm gegeben.

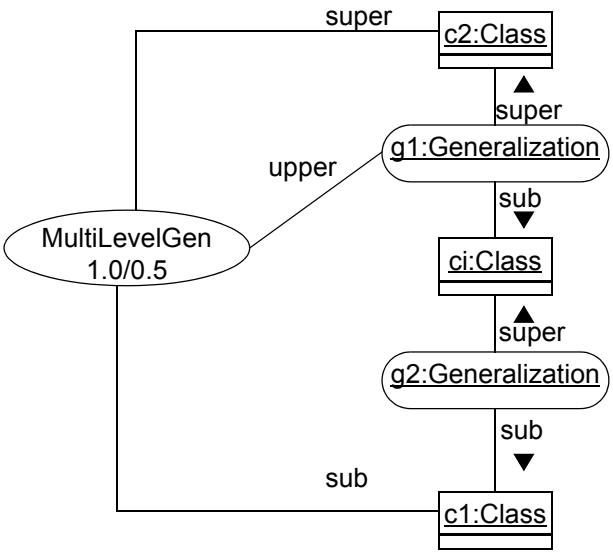


Abbildung A.16 MultiLevelGen-Regel

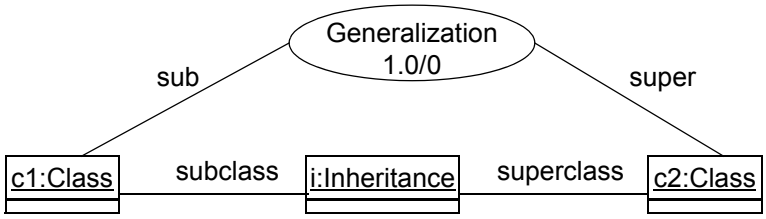


Abbildung A.17 Generalization-Regel

Auch wenn Entwurfsmuster die Softwareentwicklung einen Schritt nach vorn gebracht haben, sind die Zeiten des Spaghetti-Codes wahrscheinlich noch lange nicht zu Ende:

> *Das Visitor-Gamma-Muster ist die objektorientierte Manifestation prozeduraler Programmierung.*
>
> J. Niere

LITERATURVERZEICHNIS

[ACG+95] P.S.C. Alencar, D.D. Cowan, D.M. German, K.J.Lichtner, C.J.P. Lucena, and L.C.M. Nova. *A Formal Approach to Design Pattern Definition and Application*. Technical Report CS-95-34, University of Waterloo, Waterloo, Ontario, CA, August 1995.

[AFC98] G. Antoniol, R. Fiutem, and L. Christoforetti. *Design pattern recovery in object-oriented software*. In Proc. of the 6th International Workshop on Program Comprehension (IWPC), Ischia, Italy, pages 153–160. IEEE Computer Society Press, June 1998.

[AGG] Technical University of Berlin. *AGG, the Attributed Graph Grammar system. Online at http://www.tfs.cs.tu-berlin.de/agg (last visited June 2003)*.

[AIS+77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language*. Oxford University Press, 1977.

[Amb98] S.W. Ambler. *Process Patterns*. Cambridge University Press, 1st edition, 1998.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullmann. *Compilers – Principles, Techniques and Tools*. Addison-Wesley, 1986.

[AWT] SUN Microsystems. *AWT, the SUN Java Abstract Window Toolkit. Online at http://java.sun.com/products/jdk/awt (last visited June 2003)*.

[Bal96] H. Balzert. *Lehrbuch der Software-Technik: Software-Entwicklung*. Spektrum Akademischer Verlag GmbH, Heidelberg, Germany, 1996.

[Bas90] V.R. Basili. *Viewing Maintainance as Reuse-Oriented Software Development*. IEEE Software, 7(1):19–25, September 1990.

[BD95] A.-P. Bröhl and W. Dröschel, editors. *Das V-Modell: Standard 92*. dpunkt Verlag, Heidelberg, Germany, 2nd edition, 1995.

[Bea97] SUN Microsystems, Inc. *JavaBeans API Specification 1.01, Online at http://java.sun.com/products/javabeans/docs (last visited June 2003)*, 1997.

[BED94] J.S. Bowman, S.L. Emerson, and M. Darnovsky. *The Practical SQL Handbook - Using Structured Query Language*. Addison-Wesley, 1994.

[Ber95] V. Berzins. *Software Merging and Slicing*. IEEE Computer Society Press, 1995.

[BFLS99] S. Bünnig, P. Forbig, R. Lämmel, and N. Seemann. *A Programming Language for Design Patterns*. Technical Report 258-8/1999, University of Hagen, Hagen, Germany, September 1999.

[BH94] M. Brown and C. Harris. *Neurofuzzy Adaptive Modeling and Control*. Prentice Hall, 1994.

[Bis95] C.M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Somerlad, and M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley and Sons, Inc., 1st edition, 1996.

[Boe88] B.W. Boehm. *A Spiral Model of Software Development and Enhancement*. IEEE Computer, IEEE Computer Society Press, May 1988.

[Bro83] R. Brooks. *Towards a theory of the comprehension of computer programs*. International Journal of Man-Machine Studies, 18(2):543–554, June 1983.

[Bün99] S. Bünnig. *Entwicklung einer Sprache zur Unterstützung von Design Pattern und Implementierung eines zugehörigen Compilers (in german)*. Master's thesis, University of Rostock, 1999.

[CC90] Elliot J. Chikofsky and James H. Cross II. *Reverse Engineering and Design Recovery: A Taxonomy*. IEEE Software, 7(1):13–17, January 1990.

[CEK+00] J. Czeranski, T. Eisenbarth, H. Kienle, R. Koschke, and D. Simon. *Wiedergewinnung von Architekturinformationen: Ausblicke*. In J. Ebert and F. Lehner B. Kullbach, editors, Proceedings of the 2nd Workshop Software Reengineering (WSR 2000). published as technical report by University of Koblenz-Landau, Koblenz, Germany, 2000.

[CILS93a] D.D. Cowan, R. Ierusalimschy, C.J.P. Lucena, and T.M. Stepien. *Abstract Data Views*. SIGPLAN Journal on Research in Structured Programming, 14(1):1–13, ACM Press, January 1993.

[CILS93b] D.D. Cowan, R. Ierusalimschy, C.J.P. Lucena, and T.M. Stepien. *Application Integration: Constructing Composite Applications from Interactive Components*. Journal of Software - Practice and Experience, 23(3):255–276, John Wiley and Sons, Inc., March 1993.

[CKV96] J.O. Coplien, N.L. Kerth, and J.M. Vlissides. *Pattern Languages of Program Design (Volume 3)*. Addison-Wesley, 1996.

[CL95] D.D. Cowan and C.J.P. Lucena. *An Interface Specification Concept to Enhance Design*. IEEE Transactions on Software Engineering, 21(3):229–243, IEEE Computer Society Press, March 1995.

[CM98] V. Cherkassky and F. Mulier. *Learning From Data*. John Wiley and Sons, Inc., 1998.

[Dah95] P. Dahm. *PDL: Eine Sprache zur Beschreibung grapherzeugender Parser*. Master's thesis, University of Koblenz-Landau, Fachbereich Informatik, 1995.

[DFP+02] G.A. Di Lucca, A.R. Fasolino, F. Pace, P. Tramontana, and U. De Carlini. *Comprehending Web Applications by a Clustering Based Approach*. In Proc. of the 10th International Workshop on Program Comprehension (IWPC 2002), Paris, France, June 2002.

[DLP94] D. Dubois, J. Lang, and H. Prade. Possibilistic logic. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, Handbook of Logic in Artificial Intelligence and Logic Programming, pages 439–503. Clarendon Press, Oxford, 1994.

[EGW96] J. Ebert, R. Gimnich, and A. Winter. *Wartungsunterstützung in heterogenen Sprachumgebungen, Ein Überblick zum Projekt GUPRO*, pages 263–275. Gabler, 1996.

[EIA] Electronic Industries Association. *CDIF Transfer Format. Electronic Industries Association.* Online at http://www.cdif.org (last visited December 2002.

[EKW99] J. Ebert, B. Kullbach, and A. Winter. *Querying as an Enabling Technology in Software Reengineering*. In Proceedings of the 3rd European Conference on Software Maintenance and Reengineering (CSMR99), Amsterdam, Netherlands, pages 42–51. IEEE Computer Society Press, 1999.

[EN94] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, Redwood City, 2nd edition, 1994.

[ES89] G. Engels and W. Schäfer. *Programmentwicklungsumgebungen - Konzepte und Realisierung*, Teubner, 1989.

[Fla97] David Flanagan. *Java in a Nutshell: a desktop quick reference*. A Nutshell Handbook. O'Reilly and Associates, Inc., 981 Chestnut Street, Newton, MA 02164, 2nd edition, 1997.

[FNT98] T. Fischer, J. Niere, and L. Torunski. *Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling*. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, July 1998.

[FNTZ98] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. *Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language*. In G. Engels and G. Rozenberg, editors, Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany, LNCS 1764. Springer Verlag, 1998.

[FUZ03] *Proceedings of 12th IEEE Intl. Conf. of Fuzzy Systems (FUZZ'03). St. Louis, USA*. IEEE CS Press, May 2003.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[GK97] J.-F. Girard and R. Koschke. *Finding Components in a Hierarchy of Modules - a Step towards Architectural Understanding*. In Proceedings of the International Conference on Software Maintenance (ICSM 1997). IEEE Computer Society Press, 1997.

[Gra97] O. Gramberg. *Counting the use of software design patterns in Java AWT and NeXTstep*. Technical Report 19/1997, Department of Computer Science, University of Karlsruhe, Karlsruhe, Germany, December 1997.

[Har00] M. Harsu. *Identifying object-oriented features from procedural software.* In Proceedings of Nordic Workshop on Programming Environment Research (NWPER 2000), pages 143-158, Lillehammer, Norway, May 2000.

[HHHL03] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe. *Automatic Design Pattern Detection*. In Proc. of the 11th International Workshop on Program Comprehension (IWPC), Portland, USA (ICSE 2003 co-located), May 2003.

[HHL02] D. Heuzeroth, T. Holl, and W. Löwe. *Combining Static and Dynamic Analyses to Detect Interaction Patterns*. In Proc. of the 6th International Conference on Integrated Design and Process Technology, June 2002.

[HN90] M. T. Harandi and J. Q. Ning. *Knowledge Based Program Analysis*. IEEE Transactions on Software Engineering, 7(1):74–81, IEEE Computer Society Press, 1990.

[HNS99] U. Hausmann, M.S. Nicklous, and T. Schäck. *Smart Card Application Development Using Java*. Springer Verlag, 1999.

[IG02] I. Ivkovic and M.W. Godfrey. *Architecture Recovery of Dynamically Linked Applications: A Case Study*. In Proc. of the 10th International Workshop on Program Comprehension (IWPC 2002), Paris, France, June 2002.

[Jah99] J.H. Jahnke. *Management of Uncertainty and Inconsistency in Database Reengineering Processes*. PhD thesis, University of Paderborn, Paderborn, Germany, September 1999.

[JCC] WebGain, Inc. *JavaCC, the Java Parser Generator.* Online at http://www.experimentalstuff.com/Technologies/JavaCC/ (last visited June 2003).

[JDK] SUN Microsystems. *Java 2 Platform, java.sun.com.* Online at http://java.sun.com/j2se/ (last visited June 2003).

[JGL] ObjectSpace, Inc. *JGL, the ObjectSpace (Voyager) Java Generic Library.* Online at http://www.recursionsw.com/products/jgl/jgl.asp (last visited June 2003).

[JNW00] J.H. Jahnke, J. Niere, and J.P. Wadsack. *Automated Quality Analysis of Component Software for Embedded Systems*. In Proc. of the 8th International Workshop on Program Comprehension (IWPC), Limerick, Irland, pages 18–26. IEEE Computer Society Press, June 2000.

[JS99] J.H. Jahnke and C. Strebin. *Adaptive Tool Support for Database Reverse Engineering*. In Proc. of 1999 Conference of the North American Fuzzy Information Processing Society, New York, USA, June 1999.

[JTB] JTB. *Java Tree Builder*. http://www.cs.purdue.edu/jtb/index.html (last visited June 2003.

[JZ97] J.H. Jahnke and A. Zündorf. *Rewriting poor Design Patterns by Good Design Patterns*. In Serge Demeyer and Harald Gall, editors, Proc. of the ESEC/FSE Workshop on Object-Oriented Re-engineering. Technical Report TUV-1841-97-10, Technical University of Vienna, Information Systems Institute, Distributed Systems Group, September 1997.

[Kas96] N.K. Kasabov. *Foundations of Neural Networks, Fuzzy Systems, and Knowledge Engineering*. MIT Press, Cambridge, 1996.

[KM96] A. Konar and A. K. Mandal. *Uncertainty Management in Expert Systems Using Fuzzy Petri Nets*. IEEE Transactions on Knowledge and Data Engineering, 8(1):96–105, Academic Press, London, February 1996.

[KMS02] J. Koskinen, E. Mäkinen, and T. Systä. *Implementing a Component-Based Tool for Interactive Synthesis of UML Statechart Diagrams*. Acta Cybernetica, 15(4):547–565, 2002.

[KNNZ99] H.J. Köhler, U. Nickel, J. Niere, and A. Zündorf. *Using UML as a visual programming language*. Technical Report tr-ri-99-205, University of Paderborn, Paderborn, Germany, August 1999.

[KNNZ99b] T. Klein, U. Nickel, J. Niere, and A. Zündorf. *From UML to Java And Back Again.* Technical Report tr-ri-00-216, University of Paderborn, Paderborn, Germany, September 1999.

[KNNZ00] H.J. Köhler, U. Nickel, J. Niere, and A. Zündorf. *Integrating UML Diagrams for Production Control Systems.* In Proc. of the 22nd International Conference on Software Engineering (ICSE), Limerick, Irland, pp. 241--251, ACM Press, 2000.

[Kos92] B. Kosko. *Neural Networks and Fuzzy Systems: A Dynamical Approach to Machine Intelligence*. Prentice Hall, 1992.

[Kos00] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, University of Stuttgart, Stuttgart, Germany, October 2000.

[KP96] C. Krämer and L. Prechelt. *Design recovery by automated search for structural design patterns in object-oriented software*. In Proc. of the 3rd Working Conference on Reverse Engineering (WCRE), Monterey, CA, pages 208–215. IEEE Computer Society Press, November 1996.

[KS98] R.K. Keller and R. Schauer. *Design Components: Towards software composition at design level*. In Proc. of the 21st International Conference on Software Engineering, Los Angeles, USA, pages 302–310. IEEE Computer Society Press, April 1998.

[KSRP99] R.K. Keller, R. Schauer, S. Robitaille, and P. Page. *Pattern-Based Reverse-Engineering of Design Components*. In Proc. of the 21st International Conference on Software Engineering, Los Angeles, USA, pages 226–235. IEEE Computer Society Press, May 1999.

[KSW96] N. Kiesel, A. Schürr, and B. Westfechtel. *GRAS: A Graph-Oriented Software Engineering Database System*, pages 397–425. LNCS 1170. Springer Verlag, 1996.

[Kum92] V. Kumar. *Algorithms for Constraint Satisfaction Problems: A Survey.* AI Magazine, 13(1):32-44, 1992

[Lan99] J. Langr. *Essential Java Style: Patterns for Implementation*. Prentice Hall, 1st edition, 1999.

[Lea97] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 1997.

[Let86] S. Letovsky. *Cognitive processes in program comprehension*. In Proc. of the 1st Workshop on Empirical Studies of Programmers. Washington DC, USA, pages 58–79. Ablex Publishing Corp., Norwood, NJ, June 1986.

[Meh84] K. Mehlhorn. *Graph Algorithms and NP-Completeness*. Springer Verlag, 1st edition, 1984.

[Mey88] B. Meyer. *Eiffel: A language and environment for software engineering*. Journal of Systems and Software, 1988.

[MJS+00] H.A. Müller, J.H. Jahnke, D.B. Smith, M.A. Storey, and K. Wong. *Reverse Engineering: a roadmap*. In A. Finkelstein, editor, Future of Software Engineering. International Conference on Software Engineering (ICSE), Limerick, Irland. ACM Press, June 2000.

[MK88] H. Müller and K. Klashinsky. *Rigi - A system for programming-in-the-large*. In Proceedings of the 10th International Conference on Software Engineering (ICSE 1988), Singapore, pages 80–86. IEEE Computer Society Press, 1988.

[MOTU93] H.A. Müller, M.A. Orgun, S.R. Tilley, and J.S. Uhl. *A Reverse Engineering Approach To Subsystem Structure Identification*. Journal of Software Maintenance, 5(4):181–204, John Wiley and Sons, Inc., December 1993.

[Mül97] B. Müller. *Reengineering: Eine Einführung*. Teubner Verlag, 1997.

[NAF03] *Conference of the North American Fuzzy Information Processing Society, Chicago, USA*, July 2003.

[Nie01] J. Niere. *Using Learning Toward Automatic Reengineering*. In Proc. of the 2nd International Workshop on Living with Inconsistency (LwI), Toronto, Canada, 2001.

[Nie02] J. Niere. *Fuzzy Logic based Interactive Recovery of Software Design*. In Proc. of the 24th International Conference on Software Engineering (ICSE), Orlando, Florida, USA, pages 727–728, May 2002.

[NNWZ00] U.A. Nickel, J. Niere, J.P. Wadsack, and A. Zündorf. *Roundtrip Engineering with FUJABA*. In Proc of 2nd Workshop on Software-Reengineering (WSR), Bad Honnef, Germany (J. Ebert, B. Kullbach, and F. Lehner, eds.), Fachberichte Informatik, Universität Koblenz-Landau, August 2000.

[NNZ00] U. Nickel, J. Niere, and A. Zündorf. *Tool demonstration: The FUJABA environment*. In Proc. of the 22nd International Conference on Software Engineering (ICSE), Limerick, Irland, pages 742–745. ACM Press, 2000.

[Nov92] V. Novak. *Fuzzy logic as a basis of approximate reasoning*. In L.A. Zadeh and J. Kacprzyk, editors, Fuzzy Logic for the Management of Uncertainty, pages 247–264. John Wiley and Sons, Inc., 1992.

[NSW+02] J. Niere, W. Schäfer, J.P. Wadsack, L. Wendehals and J. Welsh. *Towards Pattern-Based Design Recovery.* In Proceedings of the 24th International Conference on Software Engineering (ICSE), Orlando, Florida, USA, pp. 338-348, May, 2002

[NW01] U.A. Nickel and R. Wagner. *Graph-Grammar Based Completion and Transformation of SDL/UML Diagrams*. In Workshop on Transformations in UML (WTUML), Genova, Italy., April 2001.

[NWW01] J. Niere, J.P. Wadsack, and L. Wendehals. *Design Pattern Recovery Based on Source Code Analysis with Fuzzy Logic*. Technical Report tr-ri-01-222, University of Paderborn, Paderborn, Germany, March 2001.

[NWW03] J. Niere, J.P. Wadsack, and L. Wendehals. *Handling Large Search Space in Pattern-Based Reverse Engineering*. In Proc. of the 11th International Workshop on Program Comprehension (IWPC), Portland, USA (ICSE 2003 co-located), May 2003.

[NWZ03] J. Niere, L. Wendehals, and A. Zündorf. *An Interactive and Scalable Approach to Design Pattern Recovery*. Technical Report tr-ri-03-236, University of Paderborn, Paderborn, Germany, February 2003.

[NZ03] J. Niere and A. Zündorf. *Reverse Engineering with Fuzzy Layered Graph Grammars*. Technical Report tr-ri-03-235, University of Paderborn, Paderborn, Germany, February 2003.

[Pal01] M. Palasdies. *Design-Pattern Spezifikation und Erkennung auf Basis von Story-Diagrammen*. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, May 2001.

[Pao89] Y.H. Pao. *Adaptive Pattern Recognition and Neural Networks*. Addison-Wesley, 1989.

[Plo] *Proceedings of the pattern language of programs conferences*. online published: http://st-www.cs.uiuc.edu/ plop/ (last visited June 2003).

[PP94] S. Paul and A. Prakash. *A Framework for Source Code Search Using Program Patterns*. IEEE Transactions on Software Engineering, 20(6):463–475, IEEE Computer Society Press, June 1994.

[Pra97] S. Prashant. *Java Beans developer's resource*. Prentice Hall, 1997.

[Pre94] W. Pree. *Design Patterns for Object-Oriented Software Development*. ACM Press, 1994.

[Qui94] A. Quilici. *A Memory-Based Approach to Recognizing Programming Plans*. Communications of the ACM, 37(5):84–93, ACM Press, May 1994.

[Rad99] A. Radermacher. *Support for Design Patterns through Graph Transformation Tools*. In Proc. of International Workshop and Symposium on Applications Of Graph Transformations With Industrial Relevance (AGTIVE), Kerkrade, The Netherlands, LNCS 1779. Springer Verlag, 1999.

[Rec01] C. Reckord. *Entwurf eines generischen Sichtenkonzeptes für die Entwicklungsumgebung Fujaba*. Bachelor's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, February 2001.

[Rip96] B.D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, 1996.

[Ris00] L. Rising. *The Pattern Almanac 2000*. Addison-Wesley, 2000.

[Ros] IBM. *Rose, the Rational Rose case tool. Online at http://www.rational.com (last visited June 2003)*.

[Roy70] W.W. Royce. *Managing the Development of Large Software Systems*. In IEEE WESCON. IEEE Computer Society Press, 1970.

[Roz99] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific, Singapore, 1999.

[RS95] J. Rekers and A. Schürr. *A Graph Grammar Approach to Graphical Parsing*. In Proc. of the IEEE Symposium on Visual Languages, Darmstadt, Germany. IEEE Computer Society Press, 1995.

[RS97] J. Rekers and A. Schürr. *Defining and Parsing Visual Languages with Layered Graph Grammars*. Visual Languages and Computing, 8(1):27–55, 1997.

[Rud97] M. Rudolf. *Concepts and Implementation of an Interpreter for Attributed Graphtransformation (in german)*. Master's thesis, Technical University Berlin, 1997.

[SK02] K. Sartipi and K. Kontogiannis. *A Graph Pattern Matching Approach to Software Architecture Recovery*. In Proc. of the 2002 IEEE International Conference on Software Maintenance (ICSM02), Montreal, Canada., pages 408–418. IEEE Computer Society Press, 2002.

[SL77] D. Steinhausen and C. Langer. *Clusteranalyse*. Walter de Gruyter, Berlin, 1977.

[Str99] C. Strebin. *Adaption unsicheren Reverse-Engineering-Wissens auf Basis konnektionistischer Methoden*. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, 1999.

[SvG98] J. Seemann and J.W. von Gudenberg. *Pattern-Based Design Recovery of Java Software*. ACM SIGSOFT Software Engineering Notes, 23(6), ACM Press, November 1998.

[SWZ95] A. Schürr, A.J. Winter, and A. Zündorf. *Graph Grammar Engineering with PROGRES*. In W. Schäfer, editor, Proc. of European Software Engineering Conference (ESEC/FSE), LNCS 989. Springer Verlag, 1995.

[TA99] P. Tonella and G. Antoniol. *Object Oriented Design Pattern Inference*. In Proc. of the 9th International Conference on Software Maintenance (ICSM), Oxford, UK., pages 230–238. IEEE Computer Society Press, September 1999.

[Tän00] G. Täntzer. *AGG: A Tool Enviroment for Algebraic Graph Transformation*. In Proceedings of Applications of Graph Transformation with Industrial Relevance (AGTIVE2000), Kerkrade, The Netherlands, LNCS. Springer Verlag, 2000.

[Tho68] K. Thompson. *Regular expression search algorithm*. Communications of the ACM, 11:419–422, 1968.

[Tog] Borland International. *Together Control Center, the Together case tool.* Online at http://www.togethersoft.com (last visited June 2003).

[TWSM94] S.R. Tilley, K. Wong, M.-A.D. Storey, and H.A. Müller. *Programmable reverse engineering*. International Journal of Software Engineering and Knowledge Engineering, pages 501–520, 1994.

[UML] OMG. *Unified Modeling Language Specification Version 1.5*. Object Management Group, 250 First Avenue, Needham, MA 02494, USA, September 2002.

[Vli98] J. Vlissides. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley, 1st edition, 1998.

[vML99] A. von Mayrhausen and S. Lang. *A coding scheme to support systemetic analysis of software comprehension*. IEEE Transactions on Software Engineering, 25(4):526–540, July 1999.

[W3C] W3C. *Jigsaw - W3C's Server*. http://www.w3.org/Jigsaw/ (last visited June 2003).

[Wag01] R. Wagner. *Realisierung eines diagrammübergreifenden Konsistenzmanagement-Systems für UML-Spezifikationen*. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, November 2001.

[WC94] C.S. Mellish W.F. Clocksin. *Programming in Prolog*. Springer Verlag, 4th edition, 1994.

[Wen01] L. Wendehals. *Cliché- und Mustererkennung auf Basis von Generic Fuzzy Reasoning Nets*. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, October 2001.

[Wen03] L. Wendehals. *Improving Design Pattern Instance Recognition by Dynamic Analysis*. In Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, USA, May 2003.

[Wer94] P.J. Werbos. *The Roots of Backpropagation: From Ordered Derivations to Neural Networks and Political Forecasting*. John Wiley and Sons, Inc., 1994.

[Wil92] L.M. Wills. *Automated program recognition by graph parsing*. PhD thesis, MIT, Cambridge, Mass., 1992.

[Wil96] L.M. Wills. *Using Attributed Flow Graph Parsing to Recognize Programs*. In Proc. of International Workshop on Graph Grammars and Their Application to Computer Science, LNCS 1073, Williamsburg, Virginia, 1994, November 1996. Springer Verlag.

[Win01] A. Winter. *Exchanging Graphs with GXL*. In Proceedings of the 9th Internation Symposium on Graph Drawing, Vienna, Austria. Springer Verlag, 2001.

[WKJ+95] M. Whitney, K. Kontogiannis, J.H. Johnson, M. Bernstein, B. Corrie, E. Merlo, J. McDaniel, R. De Mori, H. Müller, J. Mylopoulos, M. Stanley, S. Tilley, and K. Wong. *Using an Integrated Toolset for Program Understanding*. In Proceedings of the CAS Conference 1995, Toronto, Ontario, Canada, pages 262–274. The IBM centre of advanced studies, 1995.

[WKR01] A. Winter, B. Kullbach, and V. Riediger. *An Overview of the GXL Graph Exchange Language*. In International Seminar Dagstuhl Castle, Germany. Springer Verlag, 2001.

[WNGJ02] J.P. Wadsack, J. Niere, H. Giese, and J.H. Jahnke. *Towards Data Dependency Detection in Web Information Systems*. In Proc. of the Database Maintenance and Reengineering Workshop (DBMR'2002), Montreal, Canada. (ICSM 2002 Workshop), October 2002.

[Won98] K. Wong. *RIGI's Manual,* online available http://www.rigi.csc.uvic.ca/Pages/publications.html#rigipub (last visited June 2003), 1998.

[WSC96] L. Wall, R.L. Schwartz, and T. Christiansen. *Programming Perl*. O'Reilly and Associates, Inc., 2nd edition, 1996.

[YC79] E. Yourdon and L.L. Constaine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall, 1979.

[Zad65] L.A. Zadeh. *Fuzzy Sets*. Information and Control, 8:338–353, 1965.

[Zad78] L.A. Zadeh. *Fuzzy Sets as a Basis for a Theory of Possibility*. Fuzzy Sets and Systems, Elsevier Science Publishers B.V (North-Holland), 1978.

[ZSG79] M.V. Zelkowitz, A.C. Shaw, and J.D. Gannon. *Principles of Software Engineering and Design*. Prentice Hall, 1979.

[Zün95] A. Zündorf. *PROgrammierte GRaphErsetzungsSysteme*. PhD thesis, RWTH Aachen, 1995.

[Zün96] A. Zündorf. *Graph Pattern Matching in PROGRES*. In Proc. of the 5th International Workshop on Graph-Grammars and their Application to Computer Science, LNCS 1073. Springer Verlag, 1996.

[Zün01] A. Zündorf. *Rigorous Object Oriented Software Development*. University of Paderborn, 2001. (draft available online: http://www.uni-paderborn.de/fachbereich/AG/schaefer/Personen/Ehemalige/Zuendorf/AZRigSoftDraft_0_2.pdf).

INDEX